(2)

超高速プロセッサの構成方式とエミュレーションに関する研究

中　島　康　彦

１９９８年1月

目 次

図目次

表目次

Chapter 1

序論

本論文では，計算機システムにおけるプロセッサ構成方式のうち，特に，長形式命令語に基づいた超高速プロセッサの構成方式，および，動的命令変換手法によるエミュレーションに基づいたプロセッサの構成方式について述べる．

1.1 システムのコストと性能

プロセッサの構成方式は，企業の商品戦略や企業間の利害関係といった政治的要因を考慮せずに，純粋に学問的にモデル化した場合，ハードウェアとソフトウェアから構成される「システム」のコストと性能とのトレードオフによって決定される．以下に示すように，システムのコスト（Csys と定義する）は,「システムを構成する物理的素子の複雑度および量」により表すことができる．また，性能（Psys と定義する）は,「あるプログラムの実行時間の逆数」により表すことができる．

$$Csys = \text{システムを構成する物理的素子の複雑度および量} \tag{1.1}$$

$$Psys = 1/\text{あるプログラムの実行時間} \tag{1.2}$$

一方，システムを構成するハードウェアとソフトウェアのそれぞれについて，同様に学問的観点からとらえた場合，ハードウェアのコスト（Chw）とは「物理的素子の複雑度および量」である．また，性能（Phw）とは「ハードウェアを理想的に動作させるよう完全に最適化された機械命令語列を全て実行するのに要する時間の逆数」である．性能を向上させるための努力はコストを押し上げる傾向を招き，逆に，コストを抑えるための努力は何らかの性能低下につながる可能性がある．

また，機械命令語を介してハードウェアの性能を最大限に引き出すという観点からソフトウェアをとらえた場合，コンパイラやエミュレータなど，ハードウェアの駆動と密接に関連するソ

フトウェアのコスト（Csw）とは，ある言語により記述されたプログラムから機械命令語列を生成するのに要する時間，すなわち「最適化など機械命令語列の生成に要する時間」である．また，性能（Psw）とは，前述の Phw に対する，実際に生成した機械命令語列の実行時間の逆数の比，すなわち,「生成した機械命令語列の効率」である．最適化をより強化することにより，効率を 1 に近づけることができる．ただし，機械命令語列の実行時間に比べて極端に長い時間を最適化作業のために費やすことは現実的ではなく，最適化作業は時間的制約を受けることになる．また，プログラム本来の性質によっても，最適化による効率の改善度は異なる．

さて，Chw，Csw，Phw および Psw を用いて，Csys および Psys を表現することができる．まず，Csys は Chw に等しい．

$$Csys = Chw \tag{1.3}$$

次に，Psys について考える．プログラムの実行時間は，命令語列の生成に要する時間と命令語列の実行に要する時間の合計である．命令語列の実行に要する時間は，前述の定義から明らかなように Phw と Psw の積の逆数である．命令語列を生成または実行する回数に対する生成回数の比を p とすると，Psys は以下のように表すことができる．

$$Psys = 1/(p \times \text{命令語列の生成時間} + (1-p) \times \text{命令語列の実行時間}) \tag{1.4}$$

$$= 1/(p \times Csw + \frac{1-p}{Phw \times Psw}) \tag{1.5}$$

以上の結果から，システムのコストを抑えるためには，Chw を抑える必要があり，また，システムの性能を高めるためには，p の性質に応じて，以下を実現する必要があると結論づけられる．

- p を 0 に近づけられる場合，Csw を大きくしてでも Phw × Psw の項を大きくする．
- p を 0 に近づけられない場合，Csw を小さくし，かつ，Phw × Psw の項を大きくする．

1.2 プロセッサの構成方式

前節の結果をもとに，プロセッサの構成方式について考えてみる．まず，Chw を抑えるために，2 つの選択肢がある．1 つは性能を最も重要視し新規にハードウェアを開発する選択，もう 1 つはコストを重視し，アーキテクチャが異なるとしても既存のハードウェアを利用する選択である．

前者は，専用のハードウェアおよびコンパイラを構築して，プログラムから機械命令語列を生成し実行するモデルに対応する．この場合，コンパイラが動作する回数よりも機械命令語列

を実行する回数のほうがきわめて多いと仮定することができる．式 1.5における p の値をほぼ 0 と仮定すると，Psys は以下のように簡単化することができる．

$$Psys = Phw \times Psw \tag{1.6}$$

これは，Csys を抑え Psys を高めるためには，ハードウェアに関しては，Chw を抑えながら Phw を大きくし，かつ，ソフトウェアに関しては，Csw を大きくしてでも Psw を大きくする必要があることを意味する．また，式には現れない効果として，Csw を大きくすることにより，Chw を抑えられる可能性がある．具体的には，命令語のスケジューリングをコンパイラだけでなくハードウェアも行うスーパスカラ方式に対し，コンパイラが全面的にスケジューリングを行う長形式命令語方式では，Csw が大きくなる代わりに，Chw を抑えることができる．

一方，後者は，既存のハードウェア上にエミュレータを構築して，プログラム（ここでは，構成したいプロセッサの機械命令語列）から既存ハードウェアの機械命令語列を生成し，実行するモデルに対応する．この場合，一般に，エミュレータが機械命令語列を生成する回数を無視できないため，式 1.5における p の値を 0 と仮定することができない．Psys を高めるためには，Phw の大きいハードウェアを選択するとともに，p を 0 に近づける工夫が必要となる．また，p の値に応じて，Csw と Psw とのトレードオフを慎重に見きわめなければならない．

1.3 本論文の概要

著者は，2 つのプロセッサ・システムの開発プロジェクトに携わった．1 つは 1989 年から 1994 年にかけて開発した「長形式命令語に基づく VPP500 スカラプロセッサ」であり，前節において述べた専用ハードウェアおよびコンパイラを構築するモデルに相当する．もう 1 つは，1995 年から 1997 年にかけて開発した「命令エミュレーションに基づく M アーキテクチャ・プロセッサ」であり，エミュレータを構築するモデルに相当する．

まず,「長形式命令語に基づく VPP500 スカラプロセッサ」では，並列ベクトル計算機システム VPP500 のスカラプロセッサの開発を行った．

VPP500 スカラプロセッサの開発に際しては，要素プロセッサ自身の性能を飛躍的に向上させるために，当時の大型計算機のスカラプロセッサとしてはきわめて大胆な以下の手法を採り入れた．

- アーキテクチャ上の手法
  - 最大 3 操作を毎サイクル発行する 64 ビット長の長形式命令語
  - PC 相対アドレス指定による条件分岐操作

  - データ依存関係に基づき実行順序を変更できる非同期操作
  - 例外および未完了操作の複数同時表示機構

- インプリメント上の手法
  - ハードウェア量および分岐ペナルティを削減する少段数パイプライン
  - 2 個の固定小数点演算を毎サイクル発行する機構
  - 2 個の浮動小数点演算を毎サイクル発行する機構
  - 2 本のレジスタへのロードを毎サイクル発行する主記憶参照機構

VPP500 スカラプロセッサは，このような手法を適用し，また，コンパイラによる命令の最適化と連係することにより，ソフトウェアに内在する命令の並列性を最大限に引き出し，ハードウェアの並列処理に効果的に写象することのできるアーキテクチャを目指した．これらは，まさに，ハードウェアに関して Chw を抑えながら Phw を大きくし，ソフトウェアに関して Csw を大きくしてでも Psw を大きくすることを狙ったものである．

第 2章では，まず，VPP500 スカラプロセッサの開発の背景，および，根幹をなす基本思想について述べる．そして，アーキテクチャおよびインプリメントの詳細について説明し，オンキャッシュ性能を測定するベンチマーク・プログラムである Livermore-14 ループを用いて，コンパイラの最適化手法の説明および性能評価を行う．また，アーキテクチャおよびインプリメントが，コンパイラ技術とうまくかみ合っており，きわめて有効であることを明らかにする．

第 3章では，主記憶装置およびキャッシュを含めたスカラプロセッサの総合的な性能を評価するために，性能測定機構およびベンチマーク・プログラム SPECfp92 を用い，詳細な性能評価を行う．そして，評価の結果，VP2600 よりも遅いサイクルタイムであるにも関わらず，同等の性能を達成したこと，さらに高性能を達成するためには浮動小数点条件コードレジスタ数の増加が必要であること，また，直接ハードウェア量の増加につながるけれども，キャッシュ容量，ウェイ数，浮動小数点演算に要するサイクル数について改善することにより，さらに高性能を引き出せることを明らかにする．最後に，他の多くの研究者の研究成果を参照しながら，VPP500 スカラプロセッサでは達成できなかった，あるいは今後問題となるであろう以下の点について，今後の方向および可能性を探る．

- ソフトウェア・パイプライニングの追加アシスト機構
- 投機的実行のためのアーキテクチャサポート
- 分岐予測の確度向上による命令フェッチの高速化
- キャッシュ性能の改善による命令フェッチの高速化

- 参照の局所性/規則性を利用したオペランドフェッチの高速化
- レジスタの分割使用による並列度の向上
- 長形式命令語方式におけるロードモジュール互換技術

次に,「命令エミュレーションに基づく M アーキテクチャ」では，SPARC アーキテクチャに基づくプロセッサ・システムにより，M アーキテクチャのエミュレーションを行う実験システムの開発を行った．

本実験システムの最大の特徴は，オペレーティング・システムを含む全てのソフトウェアを動作させることができる点にある．このような機能は他のシステムには見られないものである．エミュレーション・システムの開発に際しては，ソフトウェア資産の最大限の利用，および，可能な限りの高性能を得るために，以下に挙げる点について特に考慮した．

- エミュレーションの対象に関して
  - オペレーティング・システムを含む全てのソフトウェア資産を動作可能とすること．
  - エミュレーションの高速化のために，ソフトウェアの修正が必要になった場合でも，修正の範囲はオペレーティング・システムに留め，アプリケーション・プログラムについては一切修正を必要としないこと．
  - M アーキテクチャにおいて無制限に許されている自己変更プログラムを正しく動作させること．また，このための性能上の不利益を定量的に評価すること．
- エミュレーション方式に関して
  - 最も高性能を実現し得る，動的命令変換手法を採用すること．
  - 専用のプロセッサを設計することは，発展性と継続性の喪失につながる．エミュレーション・システムを長期間にわたり有効とするために，現在でも年率 25%もの勢いで性能向上を継続している，汎用のマイクロプロセッサを利用すること．
  - 性能向上のために，プロセッサの記憶階層を有効に利用すること．

後述するように，動的命令変換手法の採用は，前節において述べた p の値を 0 に近づけることを狙ったものである．

第 4章では，まず，エミュレーション・システムの開発の背景，および，根幹をなす基本思想について述べる．そして，エミュレーション手法，特に動的命令変換手法の詳細について説明し，実際にベンチマーク・プログラムを走行して得た結果に基づき，詳細な性能評価を行う．実運用状態に近い I/O 頻度を有し約 40%をスーパバイザモードで走行するジョブについても，

変換後命令の再利用率は99.98%に達し，オペレーティング・システムを含めてエミュレーションを行った場合でも，動的命令変換手法がきわめて有効であることを明らかにする．一方，M命令のワーキングセットの大きさに依存してSPARCの命令キャッシュヒット率が敏感に変化すること，また，このために，変換後SPARC命令の実行に際して，命令キャッシュヒット率を向上させるための工夫がきわめて重要であることを明らかにする．

さて，本論文において述べるシステムは，

> 「ソフトウェアの時間的/空間的局所性をハードウェアに伝達し，ハードウェアがこれを有効に活用することこそが，計算機システムの性能を向上させる根源である.」

との原理に基づいて設計されている．この基本原理に基づくソフトウェアとハードウェアの密接な連係こそが，冒頭に述べたシステムのコストを抑え，性能を高めるための有効な手段である．

1.4 本研究の成果

本研究により得られた主な成果は以下の通りである．

1. 長形式命令語および非同期実行機構に基づく命令レベル並列処理方式の確立
   命令レベルの並列処理を効率良く行うための新しいアーキテクチャを提案した．具体的には，長形式命令語および非同期実行機構を導入することにより，演算器の有効利用，条件分岐命令の高速実行，命令の並列実行や追い越し処理，そして，ハードウェアの軽量化などを図った．

2. 命令レベル並列処理方式の有効性の検証
   アーキテクチャを実際にインプリメントし，各種のベンチマークプログラムを走行させた結果から，アーキテクチャおよびインプリメントが，コンパイラ技術とうまくかみ合っており，きわめて有効に機能していることを検証した．特に従来のMアーキテクチャに対する優位性を示した．一方で，さらに高性能を達成するために，浮動小数点条件コードレジスタ数，キャッシュ容量およびウェイ数，浮動小数点演算に要するサイクル数について改善することがきわめて有効であることを検証した．

3. オペレーティング・システムを含むソフトウェアの命令エミュレーション方式の確立
   Mアーキテクチャに基づく，オペレーティング・システムを含むソフトウェアの命令エミュレーション方式を確立した．特に，オーバヘッドとなる，エミュレーション対象の命令語をプラットホーム・ハードウェアの命令語に変換する部分に，高性能を得られる動

   的命令変換手法を採用し，自己変更プログラムを動作可能とするための考慮を含めた命令エミュレーション手法を確立した．

4. 命令エミュレーション方式の有効性の検証
   エミュレーション・システムを実際に構築し，オペレーティング・システムを含むプログラムを走行させた結果から，命令変換のオーバヘッドは無視できるほど小さく，また，変換後命令の蓄積の効果が非常に大きいこと，すなわち，動的命令変換手法がきわめて有効であることを検証した．さらに，動的命令変換手法では，ワーキングセットが大きい変換後命令列を実行する際に，命令キャッシュミスによる性能低下をいかに抑えるかがきわめて重要であることを検証した．

Chapter 2

長形式命令語に基づく VPP500 スカラプロセッサの構成方式

VPP500 スカラプロセッサに関し，まずアーキテクチャの特徴である，(1) 最大 3 操作を毎サイクル発行する 64 ビット長の長形式命令語；(2) PC 相対アドレス指定による条件分岐操作；(3) データ依存関係に基づき実行順序を変更できる非同期操作；(4) 例外および未完了操作の複数同時表示機構；について述べる．次にインプリメントの特徴である，(1) ハードウェア量および分岐ペナルティを削減する少段数パイプライン；(2) 2 個の固定小数点演算を毎サイクル行う機構；(3) 2 個の浮動小数点演算を毎サイクル行う機構；(4) 2 本のレジスタへの読み出しを毎サイクル行う主記憶参照機構；について述べる．最後に，Livermore-14 ループを用いた実測結果に基づき，VP2600 スカラプロセッサと比較したオンキャッシュ性能について考察を行う．

2.1 はじめに

VPP500 は，高性能スカラプロセッサに各々ベクトル機構を付加した並列ベクトル計算機システムである（図 2.1，図 2.2）．本システムの特徴は，(1) 主記憶分散配置型の並列プロセッサ構成により大容量主記憶および強力な主記憶データ供給能力を実現している点；(2) クロスバーネットワークを採用したとえばプロセッサ間に分散配置した行列を転置する際に必要なプロセッサ間一斉通信を高速化している点；(3) 個々のプロセッサを高性能ベクトルプロセッサとすることにより高い実効並列処理性能を実現している点；である [6, 7, 8, 9].

図 2.3に示すように，ハードウェアはシステム制御を行う CP （Control processor），演算処理を行う PE（Processing element），これらを相互接続するクロスバーネットワークから構成される．PE はスカラプロセッサ，ベクトルユニット，データ転送ユニットおよび主記憶から構成される．CP はベクトルユニットの代わりにグローバルシステムプロセッサとの結合機構を

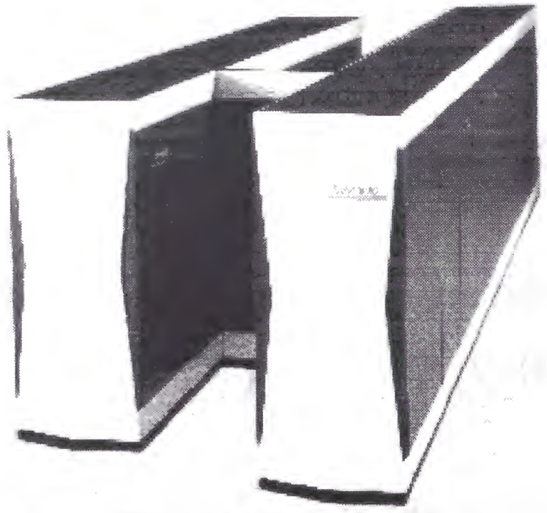


図 2.1: VPP500 システム概観

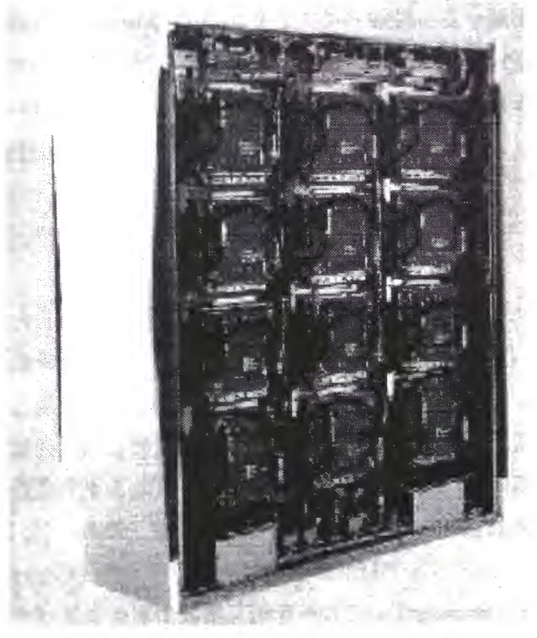


図 2.2: VPP500 システム概観（プロセッサ筐体）

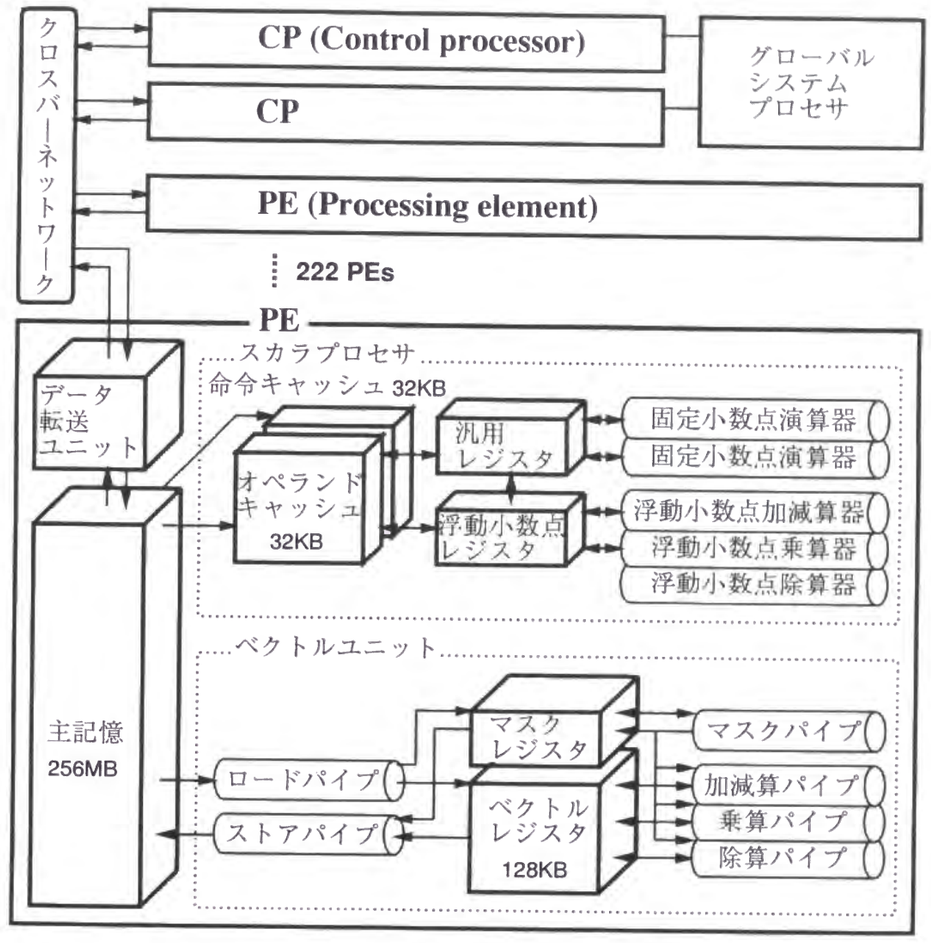


図 2.3: VPP500 システム構成

有する．本章では VPP500 スカラプロセッサの特徴について詳述する．

さて，従来の VP シリーズベクトル計算機 [4] は，M シリーズ計算機と同じスカラプロセッサ・アーキテクチャ[5] を採用してきた．長い歴史を有するこの M アーキテクチャは，以下のような特徴を有している．

- 逐次実行型のプログラミングモデルである．ソフトウェアが制御可能な並列性は，命令語レベルには存在しない．
- 分岐命令における分岐先アドレスが，レジスタ相対アドレスにより指定される．このため，レジスタの内容が確定してはじめて，分岐先アドレスが判明する．
- 命令の実行順序が保証される．後続の命令は，先行する命令の実行が完了するまで，実行してはならない．
- 割り込みがプリサイス [14] である．割り込み発生時に，PC（Program Counter）が，割り込みの原因となった命令語を直接指し示している．

ソフトウェアの観点から見た場合，これらの規定は，プログラムの可読性を高め，プログラム中に存在する問題点の再現性を良くし，また，問題となる箇所の特定を容易にするといった，プログラムの開発や検証を容易にするものである．ところが，ハードウェアの観点からすると，これらの規定は, 以下に挙げるように，インプリメントの工夫により高性能を達成する際の大きな足枷となっている．

- 一般に，ハードウェアは，命令語をパイプライン処理するために，複数の演算器を内蔵している．たとえば，演算命令の実行に使用する演算器とアドレス計算に使用する演算器は，同程度のハードウェア物量から構成される．しかし，アドレス計算に使用する演算器を通常の演算命令の実行に使用することはないため，使用効率は比較的低い．もし，このような使用効率の低い演算器をソフトウェアに対して公開することができれば，ハードウェアコストを増加することなく，性能を向上することができるはずである．
- レジスタ相対アドレスによる条件分岐では，レジスタの内容が確定するまで分岐先アドレスが不明であるため，分岐先命令をあらかじめフェッチしておくことができない．このため，条件分岐を頻繁に行うような命令列を実行する場合，パイプラインが頻繁にインタロックされることになる．
- 命令の実行順序を保証するためには，先行命令の実行が完了してから後続命令の実行を開始するような，逐次実行型の低速なハードウェア構成とするか，または，先行命令の実行が何らかの原因で遅れた場合に，後続命令の実行を待たせるための複雑なインタロック機構を有するパイプライン構成を採用する必要がある．いずれの場合でもハードウェアのコストパフォーマンスは低いものとなる．
- プリサイス割り込みを実現するためには，先行命令が正常完了しなかった場合に，後続命令を実行しなかったことにするための，複雑なハードウェア機構を用意することが必要となる．

このように，M アーキテクチャは，演算器の有効利用，条件分岐命令の高速実行，命令の並列実行や追い越し処理，そして，ハードウェアの軽量化など，インプリメントの工夫による高速化の可能性を大きく妨げている．今後，スカラプロセッサの性能を飛躍的に向上させるためには，このような高速化の可能性を追求し，ソフトウェアに内在する命令の並列性を最大限に引き出し，ハードウェアの高速処理や並列処理に効果的に写象することのできるアーキテクチャを導入しなければならない．

以上のような背景を基に，我々は，新しいアーキテクチャを採用した VPP500 スカラプロセッサを開発した．本アーキテクチャの主な特徴を以下に列挙する．

1. 1 命令語当り 1 個のベクトル操作または 1~3 個のスカラ操作を一度に発行可能とする，64 ビット長の長形式命令語（Long Instruction Word）方式を採用することにより，スーパスカラ方式において必要とされる，並列実行可能な操作を検出するためのハードウェア機構を不要とし，また，超長形式命令語（Very Long Instruction Word）方式の一般的な短所である，命令サイズ増大を抑えたこと．
2. 条件分岐操作における分岐先アドレスの指定方法を PC 相対アドレスのみとすることにより，分岐先アドレスの計算および分岐先命令のプリフェッチを高速に行うインプリメントを可能としたこと．
3. 非同期操作と呼ぶ，実行に複数サイクルを要する固定小数点乗算操作，浮動小数点演算操作，主記憶参照操作，および，ベクトル操作について，操作を並列実行する際の妨げとなる条件コード更新を必要最小限に抑えるとともに，先行する非同期操作の実行完了を待たずに後続命令を発行可能とする，非同期実行機構を設け，さらに，データ依存関係を崩さない範囲内において，非同期操作の実行順序をハードウェアが変更できる規定としたこと．
4. 例外を検出した際には，実行中の操作のうち可能なものは実行を完了し，例外を検出した操作との間にデータ依存関係があるために実行不可能なものは，未実行状態とした上で，例外を検出した操作の個々の命令語アドレスが不明であっても，ソフトウェアが例

外処理および例外処理からの復帰を正しく行うことができるよう，ハードウェアが複数の例外および未実行状態に関する十分な情報をソフトウェアに通知する機構としたこと．

次に，本アーキテクチャを最大限に利用したインプリメントの主な特徴を以下に列挙する．

1. 固定小数点演算操作および分岐操作のためのパイプラインを 3 段と短くし，ハードウェア量を抑えるとともに，高速な分岐先アドレス計算機構，および，次アドレスまたは分岐先アドレスから毎サイクル 2 命令をプリフェッチする機構を装備することにより，分岐ペナルティをほぼ 0 としたこと．

2. 各々独立した条件コードレジスタを有する 2 個の固定小数点演算器を装備し，1 命令語中に記述された 2 個の固定小数点演算操作を毎サイクル実行可能としたこと．

3. 各々1 個の浮動小数点加減算器，乗算器，除算器を装備し，特にパイプライン化された加減算器および乗算器により，1 命令語中に記述された加減算操作と乗算操作の組を毎サイクル発行可能としたこと．

4. 主記憶の連続アドレスから，一度に 2 本の 32 ビット汎用レジスタまたは 2 本の 64 ビット浮動小数点レジスタへデータを読み出すことができる主記憶参照機構を装備することにより，毎サイクル 2 個の主記憶オペランドを読み出し可能としたこと．

本章では，まず 2.2節において，アーキテクチャの特徴について述べ，続く 2.3節において，インプリメントの特徴について述べる．そして，2.4節における Livermore14 ループを用いた実測結果に基づき，2.5節において，VP2600 スカラプロセッサと比較したオンキャッシュ性能について考察を行う．

2.2 アーキテクチャの特徴

2.2.1 概要

VPP500 スカラプロセッサは，ゼロレジスタ（値を 0 に固定したレジスタ）を含む 33 本の 32 ビット汎用レジスタ（レジスタ 1 は，ユーザ・モード用とスーパバイザ・モード用に各々1 本を用意している），ゼロレジスタを含む 32 本の 64 ビット浮動小数点レジスタ，内蔵 TLB および内蔵キャッシュを有し，32 ビット固定小数点演算，IEEE754 準拠 [3] の 32/64 ビット浮動小数点演算，32 ビット仮想アドレス空間，32 ビット実アドレス空間，および，コプロセッサインタフェースを提供する汎用プロセッサである．また，主記憶空間とは別に，1024 本分のスカラ制御レジスタ空間，8192 本分のシステム制御レジスタ空間を有しており，制御レジスタを主記憶空間に割り付けた場合に生じる，アドレス変換のオーバヘッドおよび TLB の消費を回避している．

2.2.2 命令語

長形式命令語方式の採用

複数の操作を並列に実行する代表的手法に，スーパスカラ方式と長形式命令語方式がある．異なる点は，前者の場合，ハードウェアが操作間のデータ依存関係を調査し，並列実行可能な操作を特定するために，相当のハードウェア量を必要とするのに対し，後者の場合，同一命令語中の操作のスケジューリングを全面的にコンパイラに任せ，ハードウェアには，同一命令語中の全操作を必ず同時に実行開始することだけが要求されることから，ハードウェアを軽量化できることである．

このような理由から，VPP500 では長形式命令語方式を採用した．さらに，ベクトル操作を含めた全命令語を固定長とすること，および，命令サイズの増大を抑えることを目的として，64 ビット長の命令語中に 1 個のベクトル操作または 1〜3 個のスカラ操作を記述する，独自の命令語形式を採用した．おおまかに，最大 2 個の固定小数点演算操作，最大 1 個の分岐操作，最大 1 個の固定小数点乗算操作，最大 2 個の浮動小数点演算操作，最大 1 個の主記憶参照操作を組み合わせて，1 命令語中に記述することが可能である．

命令語および操作の形式

図 2.4に命令語の形式，図 2.5に操作の形式をそれぞれ示す．命令語の先頭 4 ビットが命令語の形式を決定する．この値により，残る 60 ビットの領域が，20+20+20 ビット長操作，20+40 ビット長操作，60 ビット長操作のいずれであるか，また，各操作がどのグループに属するかが決まる．グループおよび各操作の先頭 3〜5 ビットが示すオペコードの組により，操作が一意

0 3	4 23	24 43	44 63
2	load,store	float add/sub/cnv	float mul/div
3	fixed op.,shift	float add/sub/cnv	float mul/div
4	fixed op.,shift	load,store,move	float add/sub/mul/div
5	fixed op.,shift	fixed op.	float add/sub/mul/div
6	fixed op.,shift	load,store,move	branch
7	fixed op.,shift	fixed op.	branch
8	load,store,move	fixed op.	
9	fixed op.,shift	fixed op.,load,store,mul	
A	float add/sub/mul/div	fixed op.,load,store,mul	
C	load,store,move,mul	call,branch,set	
D	fixed op.,shift	call,branch,set	
E	float add/sub/mul/div	call,branch,set	
F	coprocessor (vector etc.)		
B	control		

図 2.4: 命令語の形式

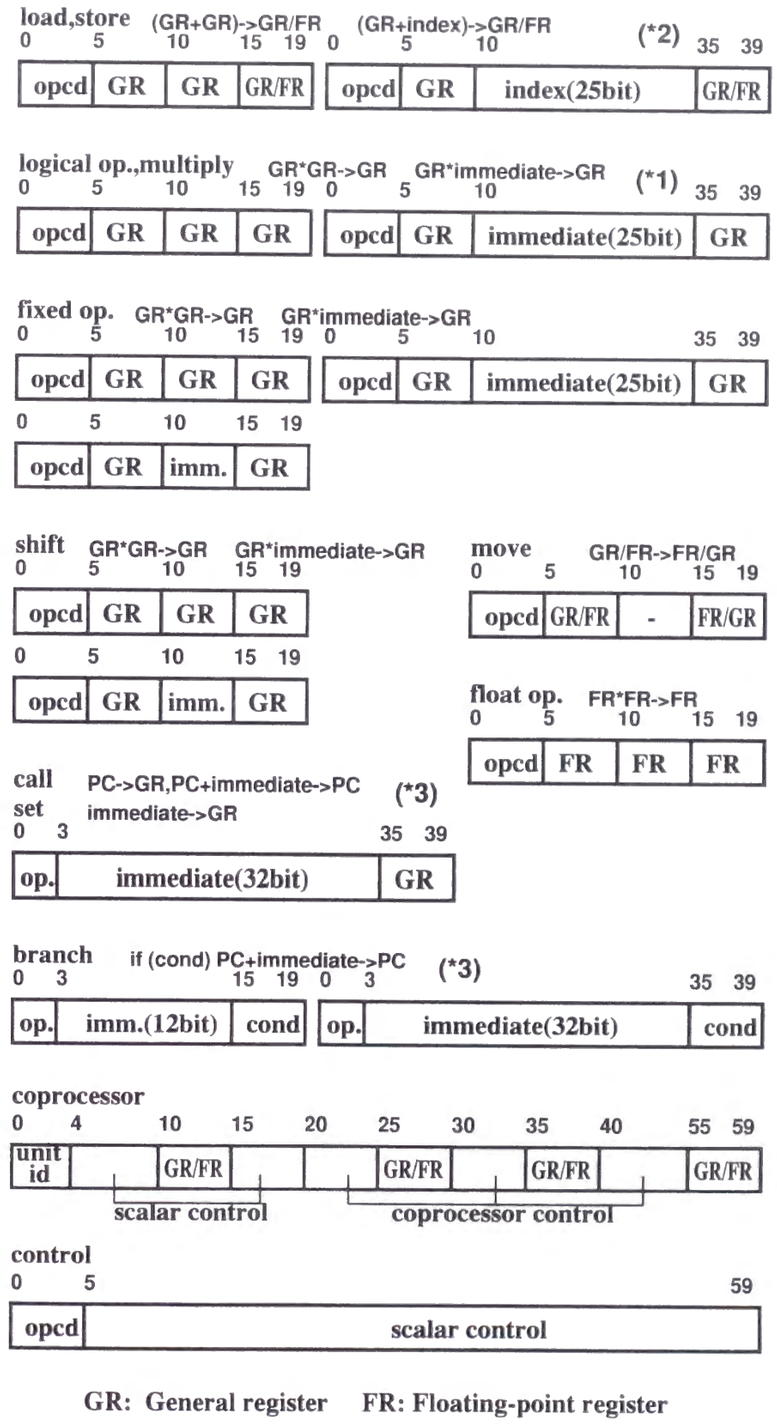


図 2.5: 操作の形式

にデコードされる．

20 ビット長操作は，3 オペランド形式を基本とし，第 2 オペランドとして 5 ビットのレジスタ番号または即値を指定可能である．40 ビット長操作は，オペランドとして 25 ビットまたは 32 ビットの即値を指定可能である．操作ごとに，出現頻度の高い即値領域ビット長が異なることを利用し，20 ビット長操作と 40 ビット長操作の両方にオペコードを用意する，または，いずれか一方に制限することにより，5 ビットのオペコードを効率よく使用している．

たとえば，図 2.5の（*1）に示すように，乗算および論理演算に使用できる即値は 25 ビット即値だけである．これは，一般的に，2 の巾乗倍を除く乗算および論理演算は，出現頻度がきわめて低く，かつ，5 ビット即値に収まる演算よりも 25 ビット即値を必要とする演算の出現頻度が高いことに基づいている．(*2) に示すように，ロード/ストア操作についても，インデックス指定には 25 ビット即値のみ使用可能である．(*3) に示す条件分岐操作および call 操作では，PC 相対アドレスにより分岐先アドレスを指定する．

2.2.3 同期操作

VPP500 では，乗算を除く固定小数点演算操作，および，分岐操作を同期操作と呼ぶ．本節では，簡単のために同期操作についてのみ説明し，非同期操作を含む正確な命令実行モデルの規定については 2.2.4節において説明する．

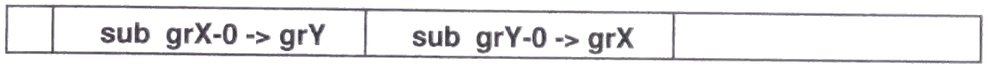
ハードウェアは，同一命令語に含まれる全ての同期操作の入力オペランドを同一命令語中の操作が更新する前に読み出す．すなわち，同一命令語中において，複数の同期操作の入力オペランドとして同一レジスタを指定することができ，また，入力オペランドとして指定したレジスタを実行結果の格納先レジスタとして指定することができる．一方，同一命令語中において，複数の同期操作の実行結果の格納先として同一レジスタを指定した場合，どの操作の実行結果が格納されるかは予測できない．

固定小数点演算操作

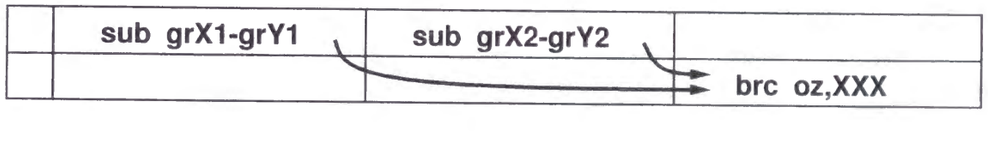
1 命令語中に，2 個の固定小数点演算操作，および，1 個の分岐操作を記述することができる．2 個の固定小数点演算器は，7 ビットの条件コードレジスタを各々1 組有しており，更新結果は，次命令の条件分岐操作に使用することができる．

記述例を図 2.6に示す．操作 “sub grX，0，grY” と操作 “sub grY，0，grX” を 1 命令語中に記述することにより，汎用レジスタ X と Y の内容を 1 命令で交換することが可能である．また，文字列処理の際に必要な NULL 検出処理のために，レジスタの内容をバイトごとに比較し，少なくとも 1 バイトが一致すれば条件コードが真になる操作を用意しており，1 命令語中に本操作を 2 個記述することにより，一度に最大 8 文字分の比較処理を行うことができる．

【ＸとＹの交換】



【Ｘ１＝Ｙ１またはＸ２＝Ｙ２の場合、分岐】



【ＮＵＬＬ文字の検出】

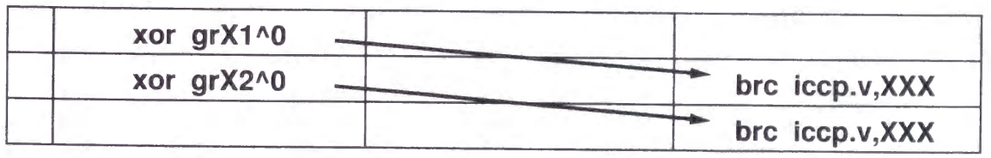


図 2.6: 同期操作の一例

分岐操作

条件分岐操作では，常に真である 1 ビット，各々7 ビットからなる 2 組の固定小数点条件コードレジスタ，2 つの零フラグの論理和を示す 1 ビット，後述する 7 ビットの浮動小数点条件コードレジスタの合計 23 ビットの中から 1 ビットを指定し，そのまま用いるかまたは反転して用いる．

たとえば，図 2.6に示すように，比較操作 “sub grX1，grY1，gr0” と “sub grX2，grY2，gr0” を 1 命令語中に記述し，2 つの零フラグの論理和を用いることにより，2 組の汎用レジスタ “X1，X2” と “Y1，Y2” を一度に比較することができる．

条件分岐操作および call 操作において指定する分岐先アドレスは，全て PC 相対アドレスである．この規定は，2.3.2節において述べるように，分岐ペナルティの削減のために大きな効果がある．

2.2.4 非同期操作と命令実行モデル

VPP500 では，実行に複数サイクルを要することが予想される，固定小数点乗算操作，浮動小数点演算操作，主記憶参照操作，および，ベクトル操作を非同期操作と呼び，先行する非同期操作の実行終了を待たずに後続命令を毎サイクル発行可能とする，非同期実行機構を規定している．非同期操作は，操作のキューイングまたは入力オペランドの読み出しを行う同期実行部分と，操作の実行結果を格納する非同期実行部分とに分けて実行される．2.2.3節において述べた同期操作は，この同期実行部分のみからなる操作である．

ここで，VPP500 の命令実行モデルについて説明する．まず，取り出した 1 つの長形式命令語は，全ての操作に関するデータの依存関係が調査され，全ての同期実行部分を実行するのに必要なデータが全てそろっている場合には，同期操作は演算器へ，また，非同期操作はキューイング機構へと送られる．これを命令の発行と呼ぶ．そして，現在の命令語の同期実行部分が全て完了したことをもって命令語の完了とし，PC を更新し，次の命令の処理を開始する．データ依存関係により，同期実行部分が完了できない操作が 1 つでも存在すれば，PC は更新されず，次の命令の処理は開始されない．すなわち，命令の発行は in-order である．

一方，非同期実行部分は，命令語の取り出しおよび同期実行部分の実行とは独立に，実行可能な時に実行される．すなわち，非同期操作が各キューから各演算器へ送られる順序および演算が完了する順序は out-of-order である．たとえば，ある命令語において汎用レジスタに対して主記憶データを読み出す操作がある場合，主記憶データの主記憶アドレスが作成され，アクセス・オペレーション・キュー（後述）にエンキューされた時点で操作の同期部分は完了し，命令も完了と見なされる．後続する命令語に含まれる全ての操作の同期実行部分において，この汎用レジスタが参照（または更新）されなければ，後続命令は，主記憶データがその汎用レジ

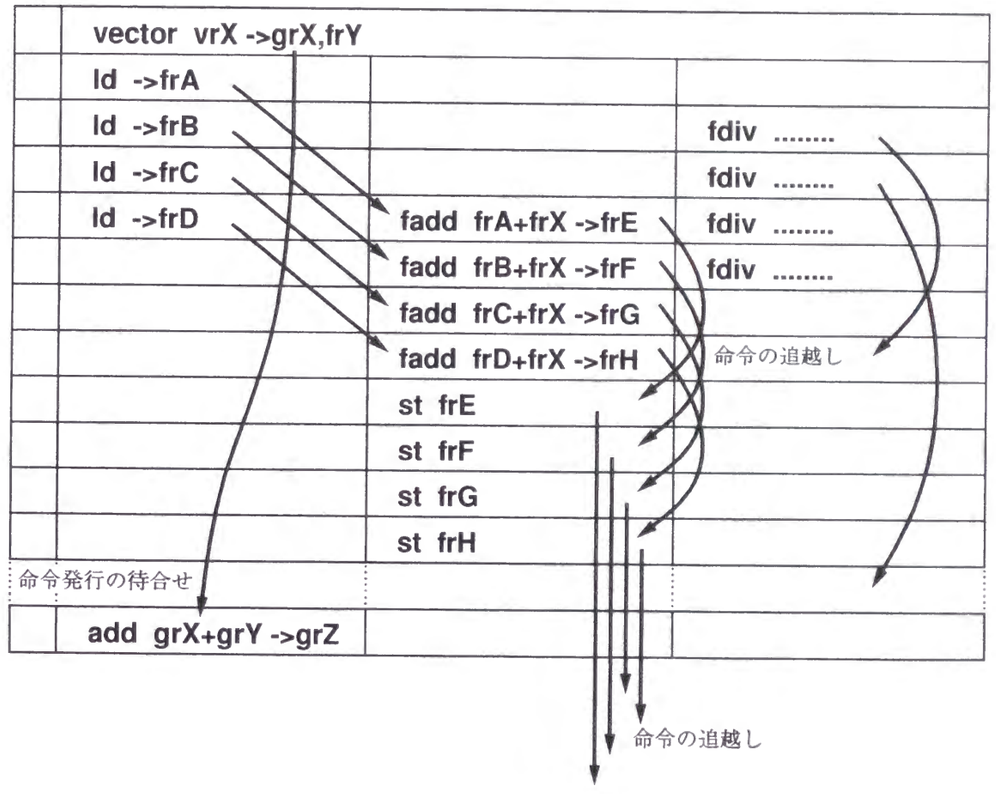


図 2.7: 非同期操作の一例

スタに格納されたかどうかに関わらず完了できる．また，後続する命令語に含まれるいずれかの操作の同期実行部分において，この汎用レジスタが参照（または更新）される場合に，まだ主記憶データが格納されていなければ，格納されるまで，後続命令に含まれる全ての操作の同期実行部分の実行が待たされる．

命令語の完了が待たされている状態をインタロックと呼ぶ．インタロックは，命令パイプライン（後述）における D ステージ，E ステージ，W ステージのいずれが止まっているかにより，大きく 3 つに分類することができる．インタロックの詳細については，3章において詳述する．

命令セット・アーキテクチャにおいて，このような命令実行モデルを導入した場合，従来のプログラミング・モデルを少し拡張して考えることが必要になる．すなわち，従来のプログラミング・モデルでは，プログラムがどこまで実行されたかを示すプログラム・カウンタの値により，計算機システム内のプログラマブルな状態（例えばレジスタの値や主記憶上のデータ）を一意に定義することができた．しかし，前述した新しい命令実行モデルでは，プログラム・カウンタの値からだけではレジスタなどの値が決定できない瞬間が存在しうる．もちろん,「プログラム・カウンタの値により計算機システムの内部状態が定義できない」というのは「プログラムできない」と同じ意味であるので,「プログラム・カウンタの値と，プログラムから可視な非同期実行部分の実行状態を表示するキューレジスタの内容から，計算機システムの内部状態が定義される」と拡張して解釈する．

このようにプログラミング・モデルが拡張されると，実際のユーザがプログラムする時に何らかの対応が必要であるように感じるかもしれない．しかし，この拡張により影響を受けるのはオペレーティングシステムのプログラマだけであり，一般のプログラマからは，命令セット・アーキテクチャにおいてデータの依存性を保証しているため，拡張された事象は観測されない（割り込みが発生した時しか，このような事象は観測されない）．また，このような拡張を行うことによって，インプリメントの自由度が増え，ハードウェア量を削減することが可能となる．この拡張に対する代償は，オペレーティングシステムの割り込みハンドラが若干複雑になることだけである．

また，この命令実行モデルでは，同一命令語中に含まれる全ての同期操作および非同期操作の入力オペランド（条件分岐操作の場合には条件コードを含む）を同一命令語中の操作または後続命令語中の操作が更新する前に読み出す．一方，同一命令語中に含まれる複数の非同期実行部分は，互いに独立に動作してよく，ハードウェアは非同期実行部分を並列に処理することができる．即ち，同一命令語中において，複数の同期操作または非同期操作の入力オペランドとして同一レジスタを指定することができ，また，入力オペランドとして指定したレジスタを実行結果の格納先レジスタとして指定することができる．ただし，同一命令語中において，複数の同期操作または非同期操作の実行結果の格納先として同一レジスタを指定した場合，どの操作の実行結果が格納されるかは予測できない．

以上のような命令実行モデルでは，図 2.7に示すように，コンパイラが，先行操作の非同期実行部分が終了するまで先行操作の実行結果を参照する新たな後続操作を発行しないように，操作のスケジューリングを行うことにより，非同期操作の見かけの実行サイクル数を同期実行部分に要するサイクル数のみ，すなわち，1 とすることができ，非同期操作を含む命令列を毎サイクル実行開始することが可能となる．なお，2.4節においてスケジューリングの具体例を示す．

例外検出時の基本動作

ところで，先行操作の完了を待たずに次々と操作を発行するためには，例外発生時の割込み処理に関して特別な考慮が必要となる．VPP500 では，命令列の実行中に例外を検出した場合，全ての非同期操作の実行終了を待ち合わせた後に割込みを発生し，1 個または複数個の例外を同時に報告する．この際，例外を検出した操作との間にデータ依存関係があるために実行できなかった操作は，未実行操作として報告する．

まず，例外の要因が，同期操作または非同期操作の同期実行部分のみに存在する場合について説明する．この時点で検出される例外は，全て，未定義命令や特権例外など，アドレスの通知のみを必要とし，オペランドの通知を必要としないものである．また，OPC（Old PC:更新前の PC の値）により，その操作のアドレスを特定することができる．従って，ハードウェアはソフトウェアに対し，例外種別および OPC だけを通知する．

次に，例外の要因が，非同期操作の非同期実行部分に存在する場合について説明する．この時点では，OPC の値が先に進んでいる可能性があり，OPC の値により操作を特定することができない．また一般に，例外を検出した操作のアドレスだけでなく，オペランドの通知を必要とする．

たとえば，ストア操作実行時にページフォルトを検出した場合，ページインを行った後にストア操作を再実行するために，ストアデータおよびストア先アドレスの通知を必要とする．もし，後続操作によるレジスタへの上書きを止める機構を設ければ，オペランドをレジスタ中に保持することができるため，操作アドレスから操作を取り出し，レジスタ番号を基に必要なオペランドを得ることができる．しかし，後続操作の実行を止める機構は，ハードウェアを複雑化するばかりでなく，後続操作の実行が遅れる要因となる．

一方，オペランドをレジスタから取り出し，他の手段により通知することができれば，後続操作によりレジスタが上書きされても構わない．VPP500 では，ハードウェアの軽量化のために，主に後者の方法を採用しており，例外を検出した，または，未実行となった，操作およびオペランドの多くを前述の制御レジスタにより通知している．通知内容の詳細については，個々に後述する．

固定小数点乗算操作

固定小数点乗算操作は，条件コードを更新せず，また演算時に例外を検出しない仕様としている．本規定により，条件コードに関する操作間の依存関係をなくし，操作の並列実行可能性を高めるとともに，非同期実行部分を処理するハードウェアの軽量化を図っている．固定小数点乗算操作は，同期実行部分において，汎用レジスタから入力オペランドの読み出しを行い，非同期実行部分において，汎用レジスタに対する乗算結果の格納を行う．

浮動小数点演算操作

浮動小数点演算操作に関しては，7 ビットの浮動小数点条件コードレジスタを 1 組規定しており，条件分岐に使用することができる．条件コードレジスタは，比較操作以外の演算操作が更新することはできない．本規定により，固定小数点乗算操作と同様，条件コードに関する操作間の依存関係を最小限とし，操作の並列実行可能性を高めている．

また，浮動小数点演算は，一部の機能をソフトウェアがシミュレートすることにより，IEEE754 の単精度/倍精度浮動小数点演算に準拠しており，ハードウェアの軽量化を図っている．たとえばハードウェアは，介入要桁あふれ例外，介入要下位桁あふれ例外，無効操作例外，零除算例外，および，不正確例外を検出する．このうち，介入要桁あふれ/下位桁あふれ例外は，マスクすることができない．ハードウェアが浮動小数点レジスタに格納した中間結果を使用して，ソフトウェアが，IEEE754 準拠の桁あふれ/ 下位桁あふれ例外をシミュレートしている．

浮動小数点演算操作は，同期実行部分において，制御レジスタ空間内の FQ（Floating-point operation queue）と呼ぶキューイング機構にエンキューされる．次に，非同期実行部分において，ハードウェアが，レジスタ依存関係に基づき，浮動小数点レジスタの読み出し，演算，演算結果の格納を行う．アーキテクチャは，レジスタ依存関係が無い場合の演算順序変更を許しており，ハードウェアは，FQ にエンキューされた操作を任意の順序で実行することができる．

浮動小数点演算例外検出時の動作

前述の例外を検出した際には，操作のオペコードおよびオペランドレジスタ番号が，FQ のエントリごとに設けた例外表示領域に表示される．例外を検出した先行操作との間にレジスタ依存関係があるために実行できない，後続の浮動小数点演算操作は，同様に，FQ 内に未実行操作として表示される．

図 2.8に，操作 “ld C” および操作 “fadd B” が，例外を検出したと仮定した場合の動作例を示す．FQ には，例外を検出した操作 “fadd B”，および，“ld C” が正常終了しなかったことにより実行できなかった操作 “fadd C” が格納され，ソフトウェアに対して通知される．

ソフトウェアに対しては，例外処理を行った後，FQ 内において未実行となった操作を FQ の

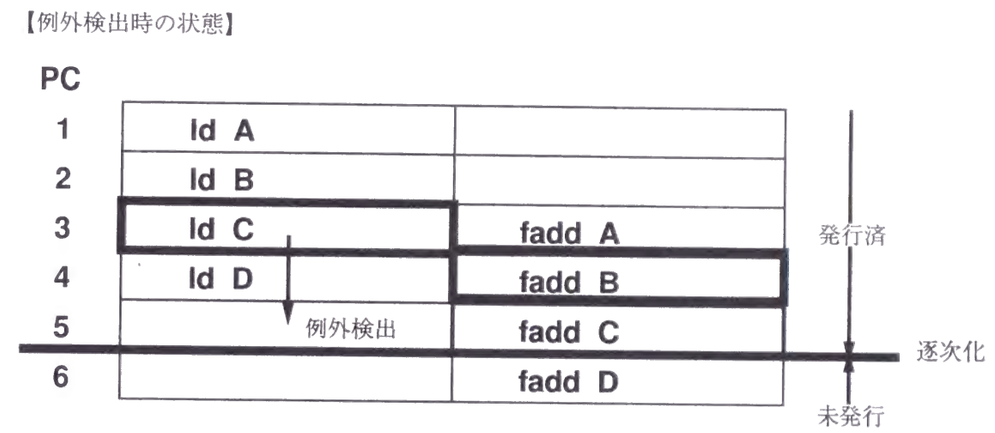


図 2.8: 例外検出時の動作

先頭から詰め直すことが要求される．ハードウェアは，再構成されたFQの内容に基づき，浮動小数点演算を再開する．

ところで前に述べた通り，オペランドを表示する際には，レジスタ番号ではなく，レジスタ内容を表示するべきである．しかし，各々64ビットの浮動小数点数を多数格納するためには，相当のハードウェア量を必要とする．ハードウェア量を抑えるためには，むしろ後続操作の実行を停止させる方法が有効であるため，浮動小数点演算例外については，レジスタ番号を表示する規定としている．

主記憶参照操作

主記憶参照操作に関しては，1命令語中に，最大2個の固定小数点演算操作，または，最大2個の浮動小数点演算操作を記述できることに対応して，主記憶の連続アドレスから，連続する2本の汎用レジスタへの8バイトロード操作，同様に，連続する2本の浮動小数点レジスタへの16バイトロード操作を規定している．

主記憶参照操作は，同期実行部分において，制御レジスタ空間内のAQ（Access operation queue）と呼ぶキューイング機構にエンキューされる．次に，非同期実行部分において，ハードウェアがアドレス変換を行い，レジスタ間および主記憶アドレス間の依存関係に基づき，主記憶参照を行う．浮動小数点演算操作と同様，アーキテクチャは，レジスタ間および主記憶アドレス間に依存関係がない場合の主記憶参照順序の変更を許しており，たとえば，先行するロード操作がキャッシュミスした場合でも，後続のロード操作がキャッシュヒットした場合には，後続のロードデータを先に使用して演算を続行することができる．すなわち，ソフトウェアによる，主記憶からキャッシュへのプリフェッチ[13]を実現可能としている．

図2.9に示すように，命令語列の空き部分に，プリフェッチのためのロード操作を埋め込むことにより，命令語数を増やすことなくプリフェッチを行うことができる．

主記憶参照例外検出時の動作

図2.8に示すように，ページフォルト等の例外を検出すると，(1) ロード操作の場合，主記憶仮想アドレス，データ長，ロードデータ格納先レジスタ番号；(2) ストア操作の場合，主記憶仮想アドレス，データ長，ストアデータ；が，AQのエントリごとに設けた例外表示領域に表示される．例外を検出した先行操作との間にデータ依存関係があるために，実行できない後続の主記憶参照操作は，同様に，AQ内に未実行操作として表示される．ソフトウェアに対しては，ページイン等の処理を行った後，例外を検出した操作，および，未実行操作を再実行することが要求される．

ベクトル操作

図2.7に示すように，ベクトル操作は，ベクトルレジスタ以外に，汎用レジスタまたは浮動小数点レジスタをオペランドとする場合がある．同期実行部分において，汎用レジスタまたは浮動小数点レジスタを読み出した後，ベクトル操作は，非同期実行部分をつかさどるベクトルユニットに対してエンキューされる．ベクトル操作において指定した汎用レジスタまたは浮動小数点レジスタに関し，先行する操作との間にレジスタ依存関係が存在する場合には，ハードウェアがベクトル操作の発行を待ち合わせる．同様に，先行するベクトル操作が，実行結果を汎用レジスタまたは浮動小数点レジスタに格納し，後続操作がこれを参照する場合，ハードウェアが，後続操作の発行を待ち合わせる．

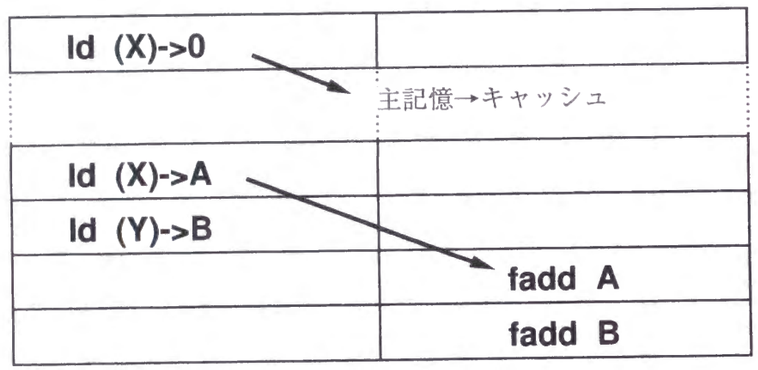


図 2.9: プリフェッチ命令の埋め込み

2.3 インプリメントの特徴

2.3.1 概要

図 2.10に，スカラプロセッサの構成を示す．スカラプロセッサは，IU（Instruction control unit），MU（Fixed-point multiply unit），FU （Floating-point unit），AU（Access control unit）と呼ぶ，各々独立に動作する 4 つのユニットから構成される．IU，MU および FU の実現には，ゲート遅延時間 60ps （Pico second）の GaAs（Gallium arsenide）素子による，ゲート数 25000 の LSI を 5 個，キャッシュを含む AU の実現には，ゲート遅延時間 70ps，RAM アクセス時間 1600ps，RAM 容量 64Kbit の ECL（Emitter coupled logic）素子による LSI を 13 個使用している（図 2.11，図 2.12）．同期操作および同期実行部分を IU が処理し，非同期実行部分をその他のユニットが処理することにより，固定小数点演算操作，固定小数点乗算操作，浮動小数点演算操作，主記憶参照操作，および，ベクトル操作を並列に実行する．

各ユニット内およびユニット間では，先行操作と後続操作の間にデータ依存関係が存在する場合にのみ，待ち合わせが生じる．後続操作が同期操作である場合は，IU が，同期操作を含む命令語全体の実行開始を待たせる．後続操作が非同期操作である場合，同期実行部分においてデータ依存関係があるときは，IU が命令語全体の実行開始を待たせ，非同期実行部分においてのみデータ依存関係があるときは，IU は同期実行部分の実行を完了し，非同期実行部分を担当する各ユニットにおいて待ち合わせが行われる．

たとえば，先行操作が主記憶からのロード操作，後続操作がロード結果を使用する同期操作（固定小数点加算操作など）である場合には，IU は後続操作を含む命令語の実行を開始せず，AU の処理完了を待ち合わせる．

次に，先行操作が主記憶からのロード操作，後続操作が浮動小数点演算操作である場合を考える．一般に，ロード操作の実行には，キャッシュミス等により多くのサイクル数を要する場合がある．ロード操作の非同期実行部分は AU，浮動小数点演算操作の非同期実行部分は FU が実行する．後続の浮動小数点演算操作が，オペランドとしてロード結果を使用する場合には，FU が AU の処理完了を待ち合わせる．一方，ロード結果を使用しない場合には，FU は AU の処理完了を待つことなく浮動小数点演算を開始する．すなわち，後者の場合，AU においてキャッシュミスが検出され，ロード操作の実行が遅延した場合においても，このために，FU における浮動小数点演算が遅延することはない．表 2.1に，主なパイプライン・インタロックによるペナルティ・サイクル数を示す．

2.3.2 命令パイプライン

命令パイプラインは 3 ステージからなり，2.3.5節において述べる，命令フェッチパイプラインと密接な関係がある．図 2.13に示すように，命令パイプラインは，操作のデコードおよび汎

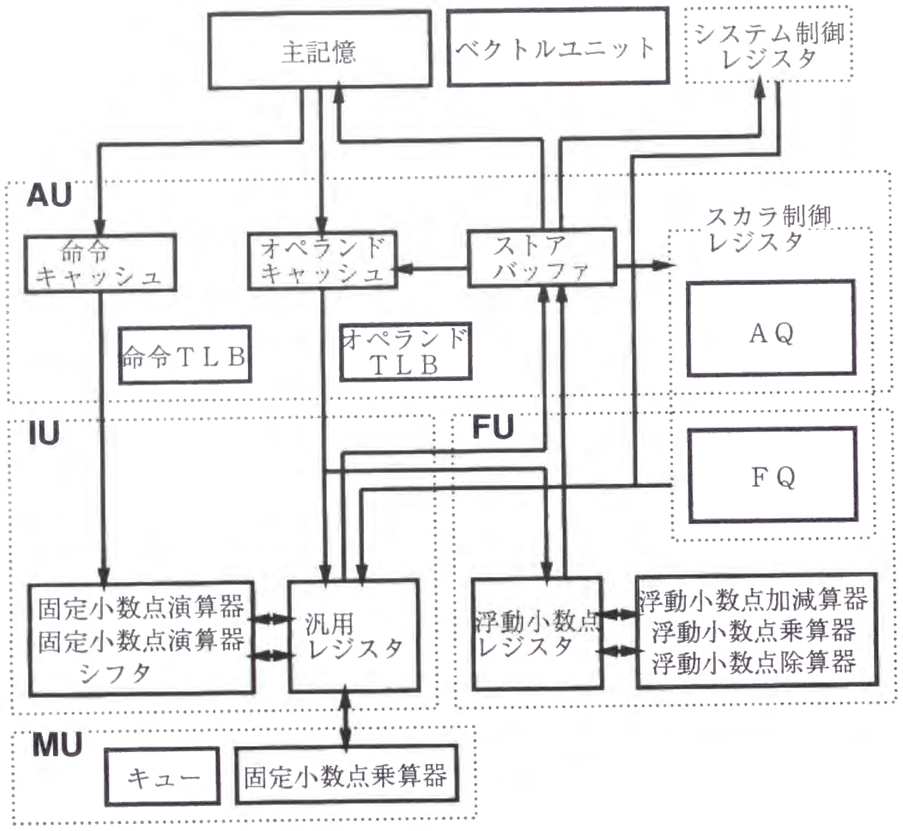


図 2.10: スカラプロセッサの構成

表 2.1: 主なペナルティ・サイクル数

操作 A の結果を操作 B が使用する時 A → B と記す	ペナルティ サイクル数
固定小数点演算→条件分岐	≃0
浮動小数点演算→条件分岐	≃3
固定小数点演算→固定小数点演算	0
固定小数点乗算→固定小数点演算	$5\sim7_{データ依存}$
浮動小数点加減乗算→浮動小数点演算	3
浮動小数点除算→浮動小数点演算	$4\sim9_{データ依存}$
ロード→固定小数点/浮動小数点演算	2

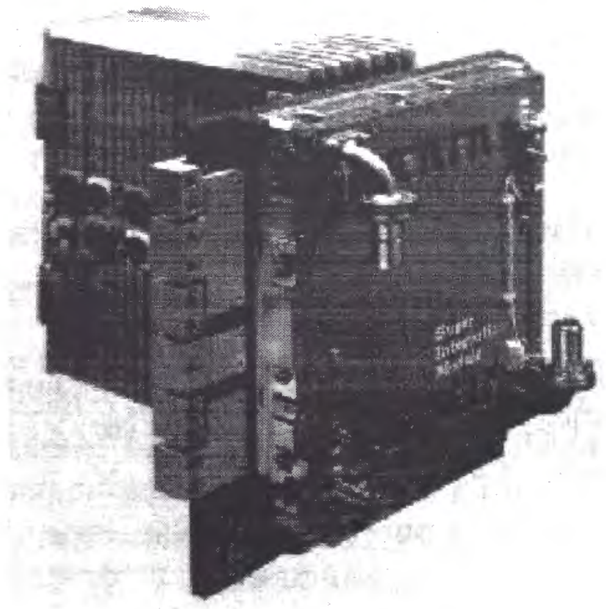


図 2.11: 要素プロセッサ概観

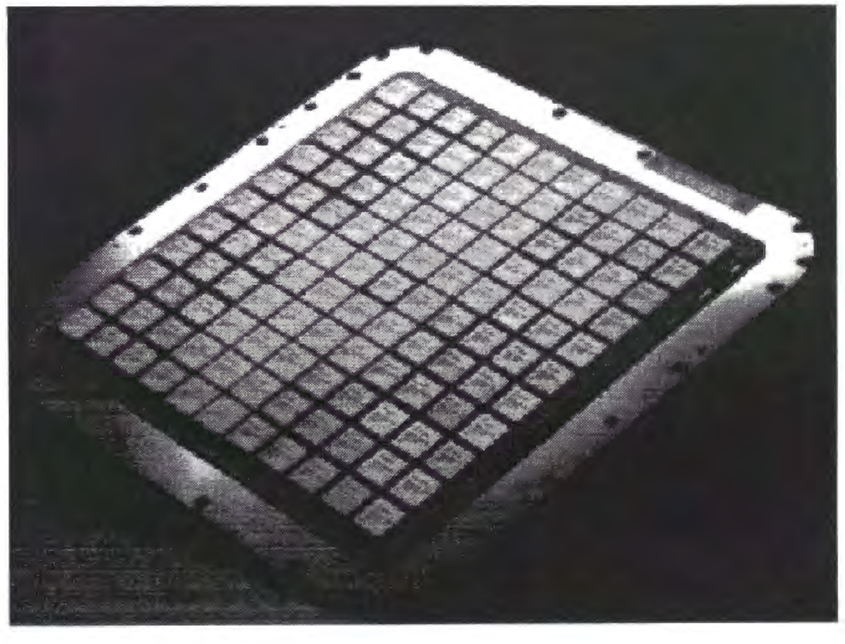


図 2.12: 要素プロセッサ概観（LSI 搭載面）

用レジスタの読み出しを行う D ステージ，演算を行う E ステージ，演算結果を汎用レジスタに格納し，条件コードおよびPC を更新する W ステージから構成される．

命令フェッチについては，以下の方式により，分岐操作のフェッチと同時に分岐先アドレス計算を行うこと，および，分岐操作のデコードと同時に分岐方向を決定することが可能となった．分岐が連続し，プリフェッチが間に合わない場合を除き，分岐時のペナルティは 0 である．

1. 各々2 命令分の，現命令バッファ，次命令バッファ，および，分岐先命令バッファを装備し，128 ビット幅のデータパスを命令キャッシュとの間に設けることにより，連続する 2 命令を毎サイクルプリフェッチする．

2. 2.2.3節において述べたように，条件分岐操作および call 操作における分岐先アドレスが常に PC 相対アドレスであることを利用して，PC と PC 相対アドレスを常に加算しておき，分岐操作のフェッチと同時に分岐先アドレスを得る．

3. 命令パイプラインを 3 ステージと短くし，分岐操作のデコード前に，直前の固定小数点演算操作が更新する条件コードを参照可能とする．

一方，サブルーチンから戻る際に使用する jmp 操作（レジスタ間接分岐）の場合，分岐先アドレスはレジスタ中に保持されているため，上記方式だけでは，jmp 操作のフェッチと同時にアドレス計算を行うことができない．jmp 操作については，以下の方式により高速化を行った．まず，VPP500 では，サブルーチン呼び出し時に call 操作を使用し，この際，汎用レジスタ GR3 に，戻りアドレスを格納するとした．次に，ハードウェアが，GR3 に格納した値を次に実行するであろう jmp 操作の分岐先予測アドレスとして，内部バッファに保持し，実際に GR3 を使用する jmp 操作を実行するまでに，GR3 に対する書き込みが行われなければ，本内部バッファの内容を分岐先アドレスとして高速に読み出すこととした．本機構により，条件分岐操作と同様，jmp 操作についても分岐先アドレスをあらかじめ知ることができる．

2.3.3 固定小数点演算機構

IU は，33 本の 32 ビット汎用レジスタ，2 個の ALU，および，1 個のバレルシフタを有する．IU はパイプライン化されており,「ALU 操作またはシフト操作」と「ALU 操作」の 2 つの操作を毎サイクル同時に実行することができる．

MU は，最大 1 個の乗算操作の非同期実行を可能とするキューイング機構，および，乗算器からなる．32 ビット × 32 ビットの乗算器は，32 ビットまたは 64 ビットの演算結果を生成することができる．なお，固定小数点乗算の使用頻度は比較的低いと予想されることから，ハードウェア量を抑えるために，パイプライン化されておらず，複数サイクルを使用して 32 ビット

× 8 ビットの演算を繰り返すことにより積を求める方式を採用している．すなわち，先行する乗算操作が終了するまで，次の乗算操作の実行が待たされる．

ただし，乗数の上位バイトほど 0 である頻度が高いことが予想されるため，乗数を表現するバイト数が少ないほど，すなわち，上位バイトから連続する 0 が多いほど，乗算に要するサイクル数が少なくなるインプリメントを行った．このため，乗算結果を次命令の入力オペランドとして使用する場合のペナルティ・サイクル数は，乗数に依存しており，5〜7 である．

2.3.4 浮動小数点演算機構

FU は，32 本の 64 ビット浮動小数点レジスタ，最大 12 個の演算操作の非同期実行を可能とする FQ，1 個の加減算器，1 個の乗算器，および，1 個の除算器からなる．加減算器では，加減算操作のほか，比較操作，浮動小数点-浮動小数点型変換操作，浮動小数点-固定小数点型変換操作を行う．除算器を除く加減乗算器がパイプライン化されており,「加減算」と「乗算」の 2 つの操作を毎サイクル同時に実行開始し，一方で，演算結果を生成することができる.「除算」については，使用頻度が比較的低いと予想されることから，ハードウェア量を抑えるために，パイプライン化されておらず，複数サイクルを使用して 2 ビットずつ結果を求める方式を採用している．すなわち，先行する除算操作が終了するまで次の除算操作の実行が待たされる．なお，操作間にレジスタ依存関係がない場合，加減乗算は，除算を追い越して演算結果を格納することができる．

図 2.14に示すように，浮動小数点演算パイプラインは，操作をデコードする Df ステージ，浮動小数点レジスタの干渉検査および読み出しを行う Ff ステージ，演算を行う E1f および E2f ステージ，演算結果を浮動小数点レジスタに格納する Wf ステージから構成される．浮動小数点演算操作は，命令パイプラインの D ステージにおいてデコードされると同時に，FQ に空きがあれば FQ にエンキューされる．FQ は，1 命令語中に記述される加減算操作と乗除算操作の組を 6 組までキューイングすることが可能である．先行操作の実行結果を待ち合わせる必要がない場合には，同一サイクルの Df ステージにおいて操作のデコードが行われる．除算は，パイプライン化されておらず，E1f および E2f ステージの代わりに，複数の Ef ステージから構成される．ただし，各 Ef ステージを半サイクル長とすることにより，除算を高速化している．

加減乗算結果を次命令の入力オペランドとして使用する場合のペナルティ・サイクル数は 3，除算結果を使用する場合は，被除数値の大小に依存しており，単精度演算の場合 4〜6，倍精度演算の場合 4〜9 である．ただし，レジスタ依存関係が FQ 内の浮動小数点演算間に閉じている場合には，浮動小数点演算パイプラインのみがインタロックし，命令パイプラインがインタロックすることはない．すなわち，FQ が満杯になるか，または，浮動小数点演算以外の操作との間にレジスタ依存関係が生じた場合を除き，命令パイプラインは動き続けることができる．

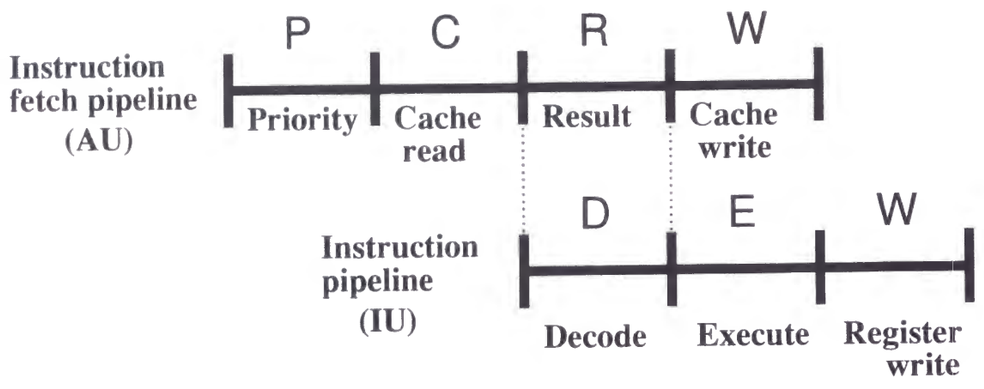


図 2.13: 命令パイプライン

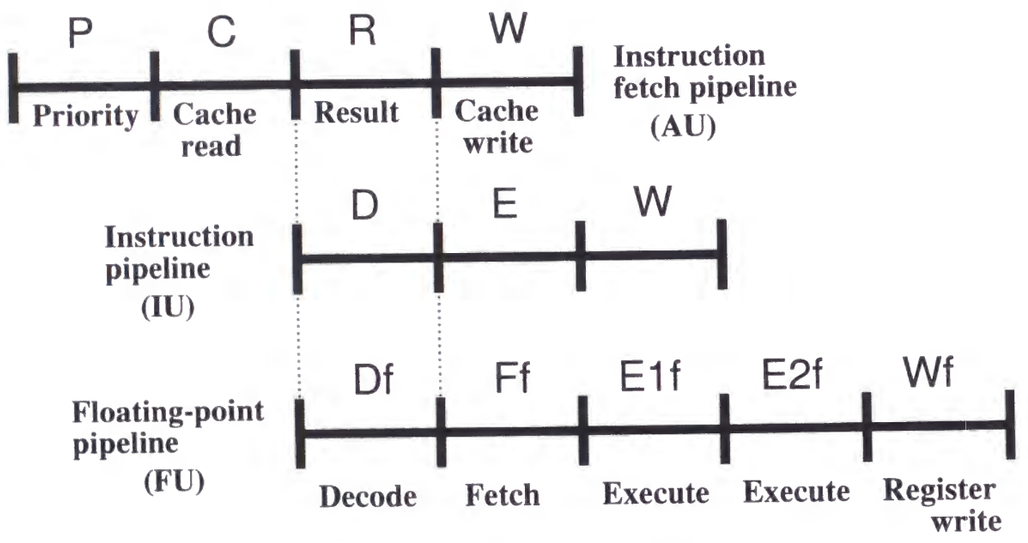


図 2.14: 浮動小数点演算パイプライン

2.3.5 主記憶参照機構

AU は，最大 2 個の主記憶参照操作の非同期実行を可能とする AQ，4 個の 8 バイトストアバッファ，TLB，および，キャッシュからなる．AU はパイプライン化されており，主記憶参照操作を毎サイクル実行開始することができる．また，データキャッシュから汎用レジスタおよび浮動小数点レジスタへは，各々レジスタ 2 本分の幅を有するデータパスを装備しており，前に述べた，2 本の汎用レジスタへの 8 バイトロード操作，および，2 本の浮動小数点レジスタへの 16 バイトロード操作を他のロード操作と同じサイクル数で実行することができる．さらに，主記憶参照操作の間にデータ依存関係がない場合，後続のロード操作が，先行するロード操作を追い越すことができる機構を装備しており，2.2.4節において述べたソフトウェアプリフェッチを可能とした．

図 2.15に示すように，命令フェッチパイプラインおよびオペランドパイプラインは，キャッシュ参照権を獲得する P ステージ，TLB およびキャッシュを参照する C ステージ，参照結果を得る R ステージ，キャッシュに書き込みを行う W ステージから構成される．P ステージでは，AU および IU からのキャッシュアクセス要求に対する優先順序付けを行う．C ステージでは，TLB を参照し仮想アドレスから実アドレスへの変換を行った後，キャッシュに登録されているかどうかを検査し，キャッシュの読み出しを行う．R ステージでは，キャッシュヒットまたはキャッシュミスの報告，例外の報告，キャッシュへの書き込みが必要な場合の優先順序付けを行う．W ステージでは，必要な場合に応じて TLB またはキャッシュへの書き込みを行う．キャッシュにヒットした場合，ロードデータを次命令の入力オペランドとして使用する場合のペナルティ・サイクル数は 2 である．

命令 TLB およびオペランド TLB には，ダイレクトマップ方式を採用している．エントリ数は各々256，ページサイズは 32K バイトである．65536 個のプロセス各々に対して 4G バイトの仮想アドレス空間を提供する動的アドレス変換機構を装備しており，多重仮想アドレス空間を実現している．TLB の各エントリは，エントリ有効ビット，プロセス識別子，論理アドレス上位 9 ビット，実アドレス上位 17 ビット，ページ制御情報（Write-protect，Read-protect，Execute-protect，Common-page，Modified-page）を保持する．

命令キャッシュおよびオペランドキャッシュには，ライトスルー方式およびダイレクトマップ方式による物理キャッシュを採用している．容量は各々32K バイト，ラインサイズは 64 バイト，キャッシュミス時のペナルティは，17 サイクルである．

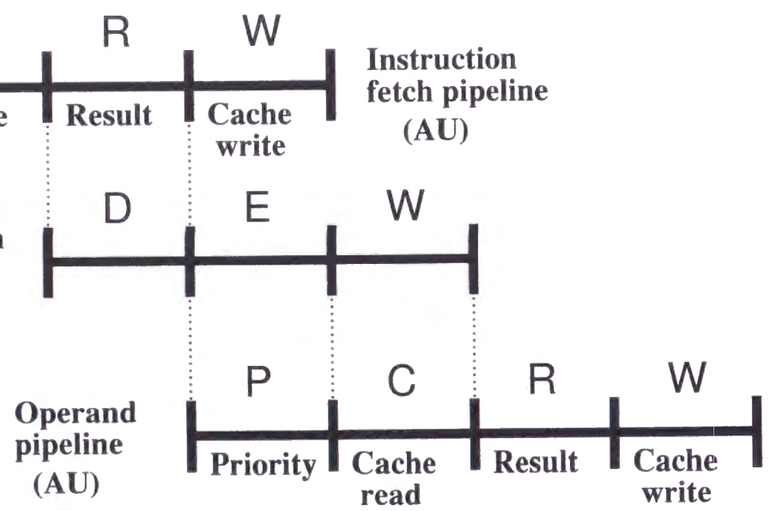


図 2.15: 命令フェッチ/オペランドパイプライン

2.4 Livermore-14 ループを用いたオンキャッシュ性能測定

VPP500 スカラプロセッサのサイクルタイムは 10ns である．最大 2 個の浮動小数点演算操作を毎サイクル発行できることから，浮動小数点演算ピーク性能は 200MFLOPS（Million floating-point operations per second）である．表 2.2に，FORTRAN コンパイラを使用し，Livermore14 ループを走行した結果を示す．なお，比較に用いた VP2600 スカラプロセッサのサイクルタイムは 6.4ns である．サイクルタイムによる正規化は行っていない．

算術平均で見ると，6.4ns の VP2600 スカラプロセッサの性能が 38.8MFLOPS であるのに対し，10ns の VPP500 スカラプロセッサの性能が 51.4MFLOPS ときわめて高い性能を発揮していることがわかる．

表 2.2: Livermore14 ループの走行結果（括弧内はサイクルタイム）

		VP2600(6.4ns)	VPP500(10ns)
Loop#	演算数	MFLOPS	MFLOPS
1	2000	47.3	135.5
2	2000	47.2	72.5
3	2000	50.1	72.8
4	1020	35.1	38.7
5	2000	42.5	20.0
6	2000	40.1	22.8
7	1920	55.1	103.6
8	1440	43.5	72.5
9	1700	48.2	73.5
10	900	32.5	21.5
11	1000	43.9	16.7
12	1000	28.1	43.5
13	896	9.5	8.7
14	1650	19.5	16.9
	算術平均	**38.8**	**51.4**

2.5 考察

本節では，Loop1 に注目して分析を行う．Loop1 に関しコンパイラは，4 重ループアンローリング，および，ソフトウェアパイプライニング [18, 19] を行っている．非同期操作である，ロード（ld）および浮動小数点加算/乗算（fadd/fmul）に関し，ロード結果は 3 サイクル後以降，演算結果は 4 サイクル後以降に使用するようにスケジューリングを行った結果，図 2.16に示す操作列が得られる．固定小数点加算（add）により，各配列に対するロード/ストアのベースアドレスを 4 要素分ずつ更新している．各操作について，“→” の左辺は入力オペランド，右辺は格納先を示す．矢印はデータ依存関係を示す．

この操作列を折り畳んだ結果，図 2.17に示す命令語列が得られる．左端の数字は，図 2.4に示した命令語の形式である．この例では，38 個の操作が「64 ビット長命令*18 個=144 バイト」に収まっており，仮にスーパスカラ方式とした場合の「32 ビット長操作*38 個=152 バイト」よりも命令サイズが小さくなっている．また，1 命令語当りの平均操作数は 2.1 個であり，命令語形式および並列実行機構が有効に使用されていると言える．

さらに，この命令語列には改良の余地がある．たとえば，[*1] の各固定小数点操作を空き領域である [**1] へ移動し，[*2] のロード操作を [**2] へ移動し，さらに [*3] の浮動小数点操作と分岐操作を 1 命令にまとめることにより，ループ当りの命令語数は 16 に減少し，1 命令語当りの平均操作数は 2.4 となる．このような緻密なスケジューリングは今後の課題である．

以上に述べたように，操作の並列実行により性能向上を図るためには，ハードウェアとコンパイラの緊密な連携が必要である．特にコンパイラの果たす役割は大きく，ハードウェアの性能を十分に引き出すためには，コンパイラに対して，ハードウェアの並列実行機構を明示することが重要である．本アーキテクチャが規定する長形式命令語および非同期実行機構は，ハードウェアを高速化するための機構であると同時に，まさに，コンパイラに対して並列実行機構を明示する機構である．

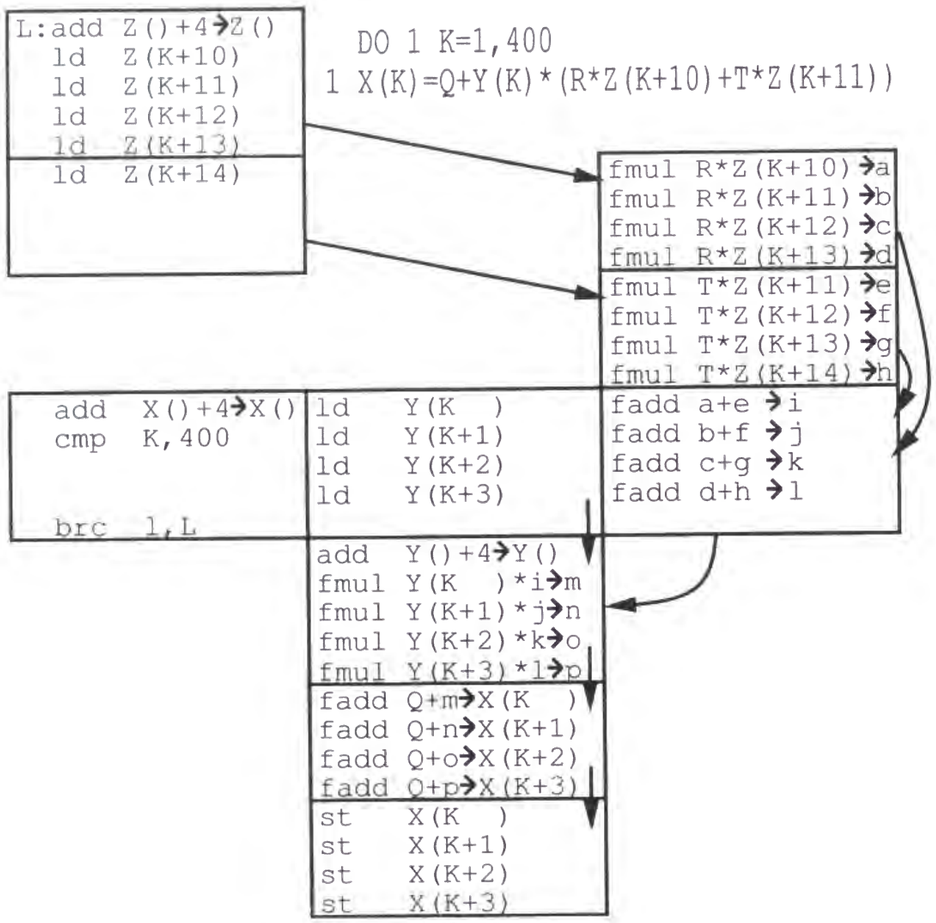


図 2.16: Loop1 の操作列

7	L:add Z()+4→Z()	add Y()+4→Y()[*1]		
2	ld Z(K+10)	fmul Y(K)*i→m		
2	ld Z(K+11)	fmul Y(K+1)*j→n		
2	ld Z(K+12)	fmul Y(K+2)*k→o		
2	ld Z(K+13)	fmul Y(K+3)*l→p		
2	ld Z(K+14)	fadd Q+m→X(K)	fmul R*Z(K+10)→a	
3	[**1]	fadd Q+n→X(K+1)	fmul R*Z(K+11)→b	
3	[**1]	fadd Q+o→X(K+2)	fmul R*Z(K+12)→c	
2	[**2]	fadd Q+p→X(K+3)	fmul R*Z(K+13)→d	
2		st X(K)	fmul T*Z(K+11)→e	
2		st X(K+1)	fmul T*Z(K+12)→f	
2		st X(K+2)	fmul T*Z(K+13)→g	
2		st X(K+3)	fmul T*Z(K+14)→h	
4	add X()+4→X()	ld Y(K)	fadd a+e →i	
4	cmp K,400	ld Y(K+1)	fadd b+f →j	
4		ld Y(K+2)	fadd c+g →k	
4		ld Y(K+3)[*2]	fadd d+h →l[*3]	
7	brc l,L[*3]			

図 2.17: Loop1 の命令語列

2.6 おわりに

本章では，VPP500 スカラプロセッサに関し，まず，アーキテクチャの特徴，および，本アーキテクチャを最大限に利用したインプリメントについて述べた．従来の M アーキテクチャに対して，大幅にアーキテクチャを改善することにより，インプリメント上の工夫による高速化の可能性を大きく広げた．具体的には，長形式命令語および非同期実行機構を導入することにより，演算器の有効利用，条件分岐命令の高速実行，命令の並列実行や追い越し処理，そして，ハードウェアの軽量化などを図った．

また，FORTRAN コンパイラを使用し，Livermore14 ループを走行した結果から，アーキテクチャおよびインプリメントが，コンパイラ技術とうまくかみ合っており，きわめて有効に機能していることを示した．

今後は，分岐操作を越えて操作を移動する広域スケジューリング機能 [20, 21, 22] を付加したコンパイラを用い，分岐操作を多く含む命令列に関して，アーキテクチャおよびインプリメントの評価を定量的に行う予定である．特に，図 2.9に示した，ゼロレジスタを格納先レジスタとするプリフェッチのためのロード操作は，格納先レジスタに関してインタロックが発生したりレジスタ内容が変更されるといった副作用がない．このため，分岐操作を越えて，比較的自由にプリフェッチ操作を移動することができると考えており，大きな効果を期待している．

Chapter 3

VPP500 スカラプロセッサの性能

VPP500 スカラプロセッサは，ハードウェア量を抑えつつ複数操作の並列実行を実現するために，3 段パイプラインを基本とする長形式命令語方式を採用した．本章では，VPP500 スカラプロセッサの性能測定項目について述べ，次に，SPECfp92[15, 16, 17] を用いた測定結果に基づき，VP2600 スカラプロセッサと比較しながら，VPP500 スカラプロセッサについて総合的な考察を行う．最後に，浮動小数点条件コードレジスタ数の増加，また，浮動小数点演算に要するサイクル数や，キャッシュ容量，ウェイ数を改善することにより，さらに高性能を引き出せることを明らかにする．

3.1 はじめに

VP シリーズベクトル計算機 [4] のスカラプロセッサは，M シリーズ計算機と同じアーキテクチャ [5] を採用してきた．しかし，このアーキテクチャは，命令実行順序や割り込み発生時の命令アドレス（Program Counter）の保証を規定しており，命令の並列実行や追い越しなど，インプリメントの工夫による高速化を妨げている．VPP500[6, 7, 8, 9] のスカラプロセッサでは，このような制約を緩和し，ソフトウェアに内在する命令の並列性をハードウェアの並列処理に効果的に写象することのできる，以下のようなアーキテクチャを採用した．

1. 最大 3 操作を同時発行可能とする，64 ビット固定長の長形式命令語（Long Instruction Word）方式を採用した．スーパスカラ方式に必要な命令スケジューリング機構を不要とし，また超長形式命令語（Very Long Instruction Word）方式の短所である命令サイズ増大を抑えた．

2. 条件分岐操作における分岐先アドレス指定を PC 相対アドレスのみとすることにより，分岐先アドレス計算，および，分岐先命令プリフェッチの高速化を可能とした．

3. 固定小数点乗算操作，浮動小数点演算操作，主記憶参照操作，および，ベクトル操作の非同期実行を可能とした．

アーキテクチャおよびインプリメントの詳細については，文献[10]または[11]を参照されたい．3.2節において，性能測定項目について述べる．続く3.3節において，SPECfp92を用いた測定結果を示し，3.4節において，測定結果に基づき，VP2600スカラプロセッサと比較しながら，VPP500スカラプロセッサの特性について考察を行う．最後に，3.5節において，さらに高性能を追求するための今後の指針について述べる．

3.2 性能測定機構による性能測定項目

（1）操作効率およびキャッシュヒット率；（2）操作頻度；（3）パイプラインインタロック率；を性能測定項目とする．操作効率から，長形式命令語方式の有効性を概観し，キャッシュヒット率，操作頻度およびパイプラインインタロック率から，プログラム毎のVPP500スカラプロセッサの挙動を把握する．

3.2.1 操作効率およびキャッシュヒット率

1命令を発行するのに要する平均サイクル数をCPI（Cycle Per Instruction），1命令あたりの平均操作数をOPI（Operation Per Instruction），1操作を発行するのに要する平均サイクル数をCPO=CPI/OPI（Cycle Per Operation）と定義する．CPIは1以上であり，パイプラインインタロックが少ないほど，理想値1に近づく．1命令語により最大3操作を発行可能であることから，OPIは1以上3以下であり，1命令にNOP（No Operation）以外の操作数が多いほど理想値3に近づく．CPOは1/3以上であり，パイプラインインタロックが少なく，かつ，命令あたりの操作数が多いほど理想値1/3に近づく．

また，毎サイクル2個の浮動小数点演算を発行できることに対して，実際に発行した浮動小数点演算の割合をFOR（Floating Operations Ratio）と定義する．平均5サイクルに2個の浮動小数点演算を発行する場合，FORは2個/（5サイクル×毎サイクル2個）=20%である．オペランドキャッシュのヒット率をOP$，命令キャッシュのヒット率をIF$と定義する．

3.2.2 操作頻度

実行した操作の出現頻度を各々，fadd/fsub（浮動小数点加減算），fcmp（同比較），fcnv（同型変換），fmul（同乗算），fdiv（同除算），load（ロード），store（ストア），etc.（固定小数点演算および分岐）と定義する．

3.2.3 パイプラインインタロック率

プログラムの実行時間に対する，インタロックによるハードウェアの停止時間の比をインタロック率と定義する．また，プログラムの実行時間に対する，インタロックが発生していない時間の比を動作率と定義する．後述する各インタロック率と動作率の関係は以下の通りである．なお定義から明らかなように，動作率の逆数が，前述したCPIに相当する．

$$\text{動作率} = 1 - (D.IF + D.FQ + D.CC + E.IP + E.FP + E.AP + W) \qquad (3.1)$$

インタロックが全く発生しない理想的なハードウェアを仮定した場合，動作率は1となり，プログラムの実行時間は，以下のようになる．

理想的なハードウェアによる実行時間 = 実際の実行時間/動作率 (3.2)

IU に関するインタロック率（D.IF）

命令フェッチパイプラインは，キャッシュ参照権を獲得する P ステージ，TLB およびキャッシュを参照する C ステージからなる．また，命令パイプラインは，命令デコードおよびレジスタ読み出しを行う D ステージ，演算を行う E ステージ，結果をレジスタに格納し，条件コード（Condition Code）および PC を更新する W ステージからなる．各 2 命令分の現命令，次命令および分岐先命令バッファに対し，次命令または分岐先命令を 128 ビット境界から毎サイクル最大 2 命令プリフェッチする．

図 3.1に，命令 i1 が CC を生成し，次命令 i2 中の条件分岐操作により，次命令 i3 または分岐先命令 t5 が実行される様子を示す．Stage0 では，2 命令 i1,2 のフェッチ Ci1,2 を行う．Stage1 では，命令 i1 のデコード Di1，次 2 命令 i3,4 のプリフェッチ Ci3,4，命令 i2 中に分岐操作を仮定した分岐先アドレス計算 Pt5,6（分岐先アドレスが PC 相対アドレスであることから PC と相対アドレスを常に加算しておく）を行う．Stage2 では，命令 i1 の演算および CC 生成 Ei1，分岐先命令 t5,6 のプリフェッチ Ct5,6，分岐操作を含む命令 i2 のデコード Di2 を行う．この時点において分岐方向が決まり，また，次命令 i3 および分岐先命令 t5 が命令バッファに存在する．Stage3 では，分岐条件に応じて，即座に，次命令 i3 または分岐先命令 t5 のデコードを開始する．

ここで，条件分岐操作における分岐先アドレス指定を PC 相対アドレスのみとした効果について説明する．たとえば，分岐先アドレスがレジスタの内容として指定されると仮定する．この場合には，Ei2 においてレジスタの内容が読み出されてはじめて分岐先アドレスが判明する．すなわち，Pt5,6 が Ei2 のステージ（Stage-3）まで遅れ，分岐先命令のデコード Dt5 が Stage-5 となることから，条件分岐に 2 サイクル余分にかかることになる．本長形式命令語方式では 2 サイクルに最大 6 操作を発行できるため，この 2 サイクルの短縮は性能向上に大きく寄与する．

次に，条件分岐が連続する場合，すなわち，分岐先命令 t5 中の条件分岐操作により，次命令 t6 または分岐先命令 u1 が実行される場合について説明する．Stage3 では，次 2 命令 t7,8 のプリフェッチ，分岐先アドレス計算 Pu1 を行う．Ct5,6 と Di2 が同時であったのに対し，分岐先命令 u1 のプリフェッチ Cu1 は Dt5 より遅れるため，この時点では分岐先命令 u1 が命令バッファに存在しない．すなわち，次命令 t6 は Stage4 から実行できるものの，分岐先命令 u1 は 1 ステージ遅れて Stage5 からの実行となる．

さて，図 3.1では，分岐先命令 t5 が 128 ビット境界の前半を占め，t5 と t6 が同時にフェッチされることから t6 の実行が遅れることはない．しかし，t5 が 128 ビット境界の後半を占める

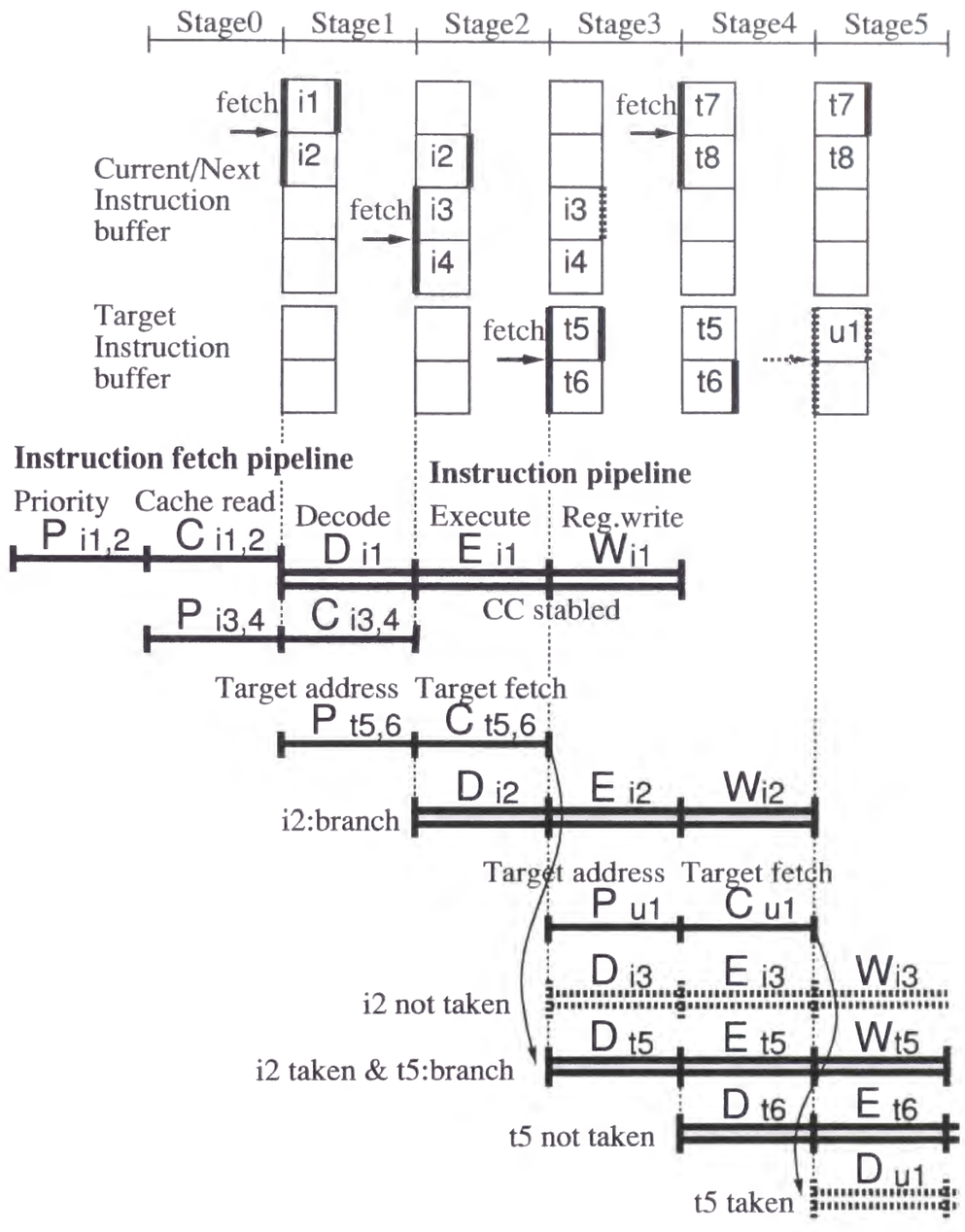


図 3.1: 命令パイプライン

場合には，t6 を新たにフェッチすることから，t6 の実行開始が遅れる．

条件分岐の連続，128 ビット境界後半への条件分岐に，命令キャッシュミスを加えて，プログラム走行時間に占める，命令フェッチに関する D ステージの遅れを D.IF と定義する．

MU に関するインタロック率（E.IP）

乗算には，乗数値に依存し 5〜7 サイクルを要する．すなわち，乗算結果を次命令が使う場合は，次命令の E ステージが乗算結果を待つ．プログラム走行時間に占める，汎用レジスタの干渉による E ステージの遅れを E.IP と定義する．E.IP は，ロード結果を次命令が使う場合の汎用レジスタの干渉（後述）も含む．

FU に関するインタロック率（D.FQ，D.CC，E.FP）

図 3.2に示すように，浮動小数点演算パイプラインは，操作をデコードする Df ステージ，浮動小数点レジスタの干渉検査および読み出しを行う Ff ステージ，演算を行う Ef ステージ，演算結果を浮動小数点レジスタに格納する Wf ステージからなる．除算の Ef ステージは半サイクル長であり，ステージ数は不定である．

浮動小数点演算操作は，命令パイプラインの D ステージにおいてデコードされると同時に，FQ に空きがあれば FQ にキューイングされる．FQ は，1 命令に記述される加減算操作と乗除算操作の組を 6 組まで格納できる．加減乗算に要するサイクル数は，IEEE-754 準拠のための演算処理が増加した分，VP2600 スカラプロセッサの 2 サイクルに対して 4 サイクルと大きい．すなわち，次命令の入力オペランドとして加減乗算結果を使う場合のインタロックは 3 サイクルである．除算結果を使う場合は，被除数値に依存し，単精度演算では 4〜6，倍精度演算では 4〜9 サイクルである．

FQ 内に存在する依存関係は，浮動小数点演算パイプラインのみのインタロックとなり，演算結果を IU が使う時（比較が生成する CC の参照，演算結果の主記憶へのストア）に，はじめて命令パイプラインのインタロックとして観測される．

プログラム走行時間に占める，FQ が満杯であることによる D ステージの遅れを D.FQ，浮動小数点比較の完了待ち（CC の確定待ち）による条件分岐操作の D ステージの遅れを D.CC，浮動小数点演算結果をストアする際のストア操作の E ステージの遅れを E.FP と定義する．

AU に関するインタロック率（E.IP，E.AP，W）

図 3.3に示すように，オペランドパイプラインは，E ステージにおけるアドレス計算結果を基にキャッシュ参照権を獲得する P ステージ，TLB およびキャッシュを参照する C ステージ，参照結果を得る R ステージ，TLB やキャッシュに書き込みを行う W ステージからなる．

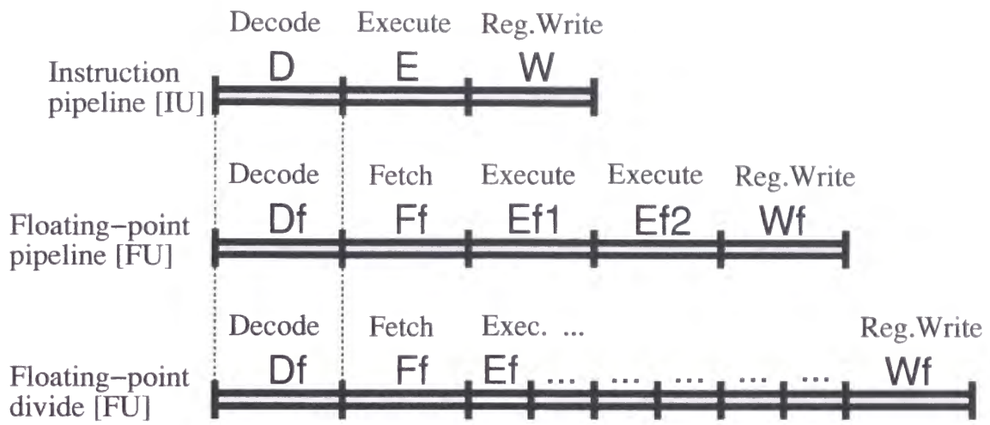


図 3.2: 浮動小数点演算パイプライン

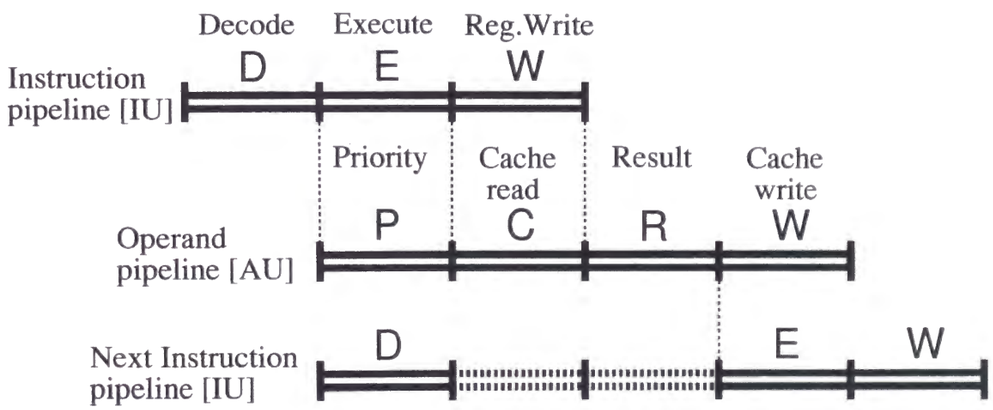


図 3.3: オペランドパイプライン

TLB およびキャッシュにヒットする時，次命令の入力オペランドとして，汎用レジスタへのロード結果を使う場合のインタロックは，2サイクルである．浮動小数点レジスタへのロード結果を浮動小数点演算が使う場合は，前節に述べたように，浮動小数点演算パイプラインのみのインタロックとなり，命令パイプラインのインタロックとしては観測されない．

主記憶参照操作は，AQ により，2個までの非同期動作が可能であり，ロード操作は，TLB およびキャッシュにヒットしている限り，また，ストア操作は，TLB にヒットしている限り，毎サイクル発行可能である．ストア操作は，キャッシュがストアスルー方式であることから，キャッシュミスによるインタロックは生じない．すなわち，TLB にヒットしている限り，先行するストア操作を後続のロード操作が追い越すことはない．

一方，非同期動作中の2個のロード操作のうち，1個がキャッシュミスを検出した場合には，3個目のロード/ストア操作の発行が待たされる．キャッシュミス時のロード操作のインタロックは，17サイクルである．

乗算結果を次命令が使う場合の汎用レジスタの干渉（前述）に加えて，プログラム走行時間に占める，ロード結果を次命令が使う場合の，汎用レジスタの干渉による E ステージの遅れを E.IP（キャッシュミスによる遅れは含まない），ロード結果を次命令が使う場合の，キャッシュミスによる E ステージの遅れを E.AP，先行ロード操作においてキャッシュミスを検出したことによる後続ロード操作の発行の遅れを W と定義する．

たとえば，ロード結果を直後の命令が使う頻度が高く，キャッシュヒット率が高いプログラムでは，E.IP は高い値，E.AP および W は低い値を示す．一方，ソフトウェアパイプライニングの適用により，ロード結果を2サイクル以上離れた命令が使うプログラムでは，E.IP は低い値を示す．この時キャッシュヒット率が低ければ E.AP が高くなり，さらにロード操作が毎サイクル発行される場合は W も高い値を示す．

3.3 SPECfp92 を用いた性能測定

FORTRAN コンパイラおよび C コンパイラを使用し，VPP500 スカラプロセッサ（サイクルタイム 10ns）上において，SPECfp92 の各プログラムを走行した結果を VP2600 スカラプロセッサ（サイクルタイム 6.4ns）に対する相対性能により，表 3.1の左段に示す．右段は，左段に 10ns/6.4ns を乗じた値であり，同一サイクルタイムを仮定した性能比を表している．幾何平均は 1.56 である．一方，プログラムにより 0.73 から 3.94 まで大きなばらつきがある．

さて，性能測定ハードウェア機構によるサンプリング調査を行って得た，各項目の測定結果を表 3.2，表 3.3，表 3.4に示し，個々のプログラムに関して分析を行う．

3.3.1 spice2g6

LU 分解（ただし，ループアンローリングおよびソフトウェアパイプライニングの効果がない）が中心である．浮動小数点演算よりも，PIVOT のためのロード，整数比較，条件分岐が多く，etc. が 56.8%と高い．典型的な，“A=配列 (B+I)” および “IF(A.LT.B)GOTO” に対する命令列を示す．各行先頭の “番号:” は，命令語の追い番を表し、命令語間の-2-は，2サイクルの命令パイプラインインタロックを表す．

```
1: load     .. 汎用レジスタへの変数 B のロード
   -2-      .. ロード待ちインタロック E.IP
2: add      .. B+I により配列の添字を求める
3: shift    .. 4 倍して要素アドレスとする
4: load     .. 配列要素からのロード
   -2-      .. ロード待ちインタロック E.IP
5: add      .. ロード結果を使用
-----------------------------------------
1: load     .. 変数 A のロード
   -2-      .. ロード待ちインタロック E.IP
2: compare  .. A と B を比較
3: branch   .. A<B なら分岐
```

このように，1操作のみからなる命令列の頻度が高いため，OPI が 1.14 と極端に悪い．また，ロード待ちインタロック E.IP が 44.0%ときわめて高いため，CPO が 2.19 と大きい．

表 3.1: VP2600 スカラプロセッサに対する性能

	対 VP2600 (6.4ns) 比	サイクルタイムにより正規化
spice2g6	0.84	1.31
doduc	0.71	▲ 1.11
mdljdp2	0.82	1.29
wave5	1.07	1.67
tomcatv	1.15	1.80
ora	0.89	1.40
alvinn	2.52	◎ 3.94
ear	1.38	◎ 2.15
mdljsp2	0.77	▲ 1.20
swm256	1.79	◎ 2.80
su2cor	0.96	1.51
hydro2d	0.83	1.29
nasa7	1.00	1.56
fpppp	0.47	▲ 0.73
幾何平均	1.00	1.56

▲は性能比が特に悪いもの，◎は特に良いものを示す．

表 3.2: 操作効率およびキャッシュヒット率

	CPI	OPI	CPO	FOR(%)	OP $ (%)	IF $ (%)
spice2g6	2.50	1.14	2.19	1.21	85.33	99.79
doduc	2.28	1.21	1.88	6.74	96.28	97.87
mdljdp2	2.42	1.17	2.06	12.18	94.95	99.99
wave5	1.80	1.41	1.28	11.88	97.33	99.77
tomcatv	2.07	1.36	1.53	12.85	91.30	99.98
ora	2.23	1.20	1.85	9.72	100.00	99.99
alvinn	1.58	1.66	0.95	19.44	96.40	99.69
ear	2.20	1.35	1.63	4.45	99.51	99.82
mdljsp2	2.32	1.13	2.04	12.11	98.75	99.99
swm256	1.41	1.73	0.81	33.19	96.27	99.99
su2cor	2.45	1.46	1.67	10.52	78.62	99.99
hydro2d	2.53	1.28	1.97	8.69	89.97	99.60
nasa7	3.33	1.54	2.16	8.24	77.34	99.96
fpppp	2.86	1.25	2.30	7.05	96.74	94.21

下線は際立つ値を示す．

表 3.3: 操作頻度（%）

	fadd fsub	fcmp	fcnv	fmul	fdiv	load	store	etc.
spice2g6	2.6	0.6	0.1	1.6	0.5	28.8	9.1	56.8
doduc	11.0	3.4	1.3	8.0	1.8	26.3	17.1	31.2
mdljdp2	22.9	11.3	0.2	14.8	1.0	15.2	9.0	25.7
wave5	12.5	1.7	1.8	13.8	0.6	21.6	14.3	33.6
tomcatv	22.7	1.2	0.0	14.7	0.6	30.5	11.7	18.5
ora	15.0	2.5	0.0	14.7	3.8	22.3	7.8	33.9
alvinn	18.2	0.1	0.7	18.1	0.0	37.0	10.3	15.5
ear	6.9	1.7	0.0	5.9	0.0	26.3	11.4	47.8
mdljsp2	22.8	11.1	0.2	14.4	1.0	16.4	8.8	25.3
swm256	32.5	0.0	0.0	20.7	0.8	28.3	11.1	6.6
su2cor	15.3	0.9	0.7	17.5	0.7	22.6	10.6	31.7
hydro2d	16.0	7.3	0.0	9.9	1.1	25.4	10.2	30.1
nasa7	17.5	0.4	0.1	17.1	0.5	32.5	13.5	18.5
fpppp	14.9	0.4	0.0	16.9	0.1	32.0	18.4	17.3

下線は際立つ値を示す.

表 3.4: インタロック率および動作率（%）

	W	E.AP	E.IP	E.FP	D.IF	D.FQ	D.CC	動作率
spice2g6	0.8	2.5	‡44.0	7.7	4.2	0.5	1.6	38.7
doduc	2.7	3.6	3.9	15.1	15.0	4.8	11.7	43.2
mdljdp2	2.7	1.9	2.1	1.8	3.9	12.0	†39.2	36.4
wave5	2.5	3.4	3.2	†20.1	4.4	1.8	7.5	57.1
tomcatv	12.9	4.9	8.3	11.4	0.8	8.8	6.5	46.4
ora	0.0	0.7	3.9	†21.8	3.8	16.5	6.5	46.8
alvinn	†23.5	2.2	1.4	2.8	4.4	2.0	0.0	63.7
ear	0.5	1.4	3.2	5.0	12.8	0.3	14.8	62.0
mdljsp2	0.3	1.3	2.1	0.9	4.9	10.4	†38.1	42.0
swm256	5.3	5.5	0.1	9.2	1.7	8.7	0.1	69.4
su2cor	†22.3	7.4	1.7	14.3	2.9	7.0	3.6	40.8
hydro2d	7.3	3.1	3.3	11.0	5.9	1.5	†29.5	38.4
nasa7	‡40.5	10.7	0.8	10.0	2.2	6.7	0.5	28.6
fpppp	1.2	8.7	5.5	†29.0	†25.1	0.3	0.0	30.2

下線は10%以上，†は20%以上，‡は40%以上を示す.

動作率は，100からインタロック率の合計を引いた値を示す.

3.3.2 doduc

浮動小数点比較に基づく条件分岐が多く，様々なサブルーチンを頻繁に渡り歩く．ループアンローリングやソフトウェアパイプライニングは困難である．典型的な，“IF(X)100,200,300”に対する命令列を示す．

```
1: f.load     .. 浮動小数点レジスタへのロード
2: f.compare  .. ロード結果Xの判定
   -5-        .. CC確定待ちインタロック D.CC
3: branch     .. 正なら分岐
4: branch     .. 零なら分岐      分岐連続 D.IF
```

CC確定待ちインタロックは，ロード待ちの-2-に，比較結果待ちの-3-を加えた，-5-となる．このような命令列を反映して，D.CCが11.7%，D.IFが15.0%と，各々が高い値を示している．多くのサブルーチンを渡り歩くことを反映して，IF$が97.87%と比較的低いことから，D.IFが高い要因として，条件分岐の連続，128ビット境界後半への条件分岐に加えて，命令キャッシュミスも挙げられる．

3.3.3 mdljdp2

fcmpが11.3%であることからもわかるように，浮動小数点演算に基づく条件分岐が非常に多い．部分的に，ループアンローリングおよびソフトウェアパイプライニングを適用することが可能である．しかし，効果は小さい．典型的な，“IF(A-B.LT.C)THEN”に対する命令列を示す．

```
1: f.load     .. 浮動小数点レジスタへのロード
2: f.subtract .. ロード結果AからBを減算
3: f.compare  .. 減算結果の判定
   -8-        .. CC確定待ちインタロック D.CC
4: branch     .. A-B>=Cなら分岐
```

CC確定待ちインタロックは，ロード待ちの-2-に，減算待ちの-3-および比較結果待ちの-3-を加えた，-8-となる．このような命令列を反映して，D.CCが39.2%と高い値を示している．

3.3.4 wave5

他のプログラムに比べて，fcnvが1.8%と型変換が多いのが特徴である．部分的に，ループアンローリングおよびソフトウェアパイプライニングを適用することが可能であるが，効果は小さい．例として，“AX=BX*X(I)+C;AY=BY*Y(I)+C”に対する命令列を示す．


```
1: f.load     .. X(I)のロード
2: f.load     .. Y(I)のロード
3: f.multiply .. BX*の乗算
4: f.multiply .. BY*の乗算
5: f.add      .. +Cの加算
6: f.add      .. +Cの加算
7: f.convert  .. 整数AXへの型変換
   -8-        .. 結果待ちインタロック E.FP
8: move       .. AXを汎用レジスタへ移動
```

2つのステートメントに対する命令が交互に発行され，依存関係が1命令おきとなっているため，インタロックが軽減されている．結果待ちインタロックは，ロード待ちの-1-，乗算結果待ちの-2-，加算結果待ちの-2-，型変換待ちの-3-を合計した，-8-である．このようなインタロックが多いことから，E.FPは20.1%と高い．

3.3.5 tomcatv

DOループのループアンローリングおよびソフトウェアパイプライニングが可能であり，レジスタ干渉の少ない命令列を生成することが可能である．例として，“A(I)=B(I)-C(I)*D”を8重アンローリングし，ソフトウェアパイプライニングを適用した命令列の一部を示す．

```
 1: f.load .. C(I)のロード×8個
 2: f.load    および
 3: f.load .. B(I)のロード×8個
 4: f.load  f.multiply .. *Dの乗算×8個
 5: f.load  f.multiply
 6: f.load  f.multiply
 7: f.load  f.multiply
 8: f.load  f.multiply
 9: f.load  f.multiply
10: f.load  f.multiply  f.subtract .. B(I)-
11: f.load  f.multiply  f.subtract    の減算
12: f.load              f.subtract    ×8個
13: f.load              f.subtract
14: f.load              f.subtract
15: f.load              f.subtract
```

```
16: f.load              f.subtract
17: f.store             f.subtract
18: f.store .. ストア×8個
    (以下省略)
```

ただし，ロードが連続して発行される一方，キャッシュヒット率OP$が91.30%と低いため，Wが12.9%と高い値を示している．

3.3.6 ora

他のプログラムに比べて，fdivが3.8%と，除算が多いのが特徴である．ループアンローリングおよびソフトウェアパイプライニングの適用は困難である．wave5に似て，依存関係のある浮動小数点演算が連続して発行されるため，E.FPが21.8%と高い．倍精度除算は，最大9サイクルを要し，FQに長時間留まるため，wave5と異なり，FQ待ちインタロックのD.FQが16.5%と高い．

3.3.7 alvinn

tomcatvと同様，DOループのループアンローリングおよびソフトウェアパイプライニングにより高い性能を引き出せる．Wが23.5%と目立つのは，W以外のインタロック率がきわめて低いためである．OPIが1.66，CPOが0.95，FORが19.44%といずれも良い値を示し，正規化後の対VP2600性能比でも3.94と最も良い値を示している．

3.3.8 ear

alvinnと同様，DOループのループアンローリングが可能である．しかし，主なループ中に，浮動小数点比較に基づく条件分岐や絶対値演算があるのに対し，VPP500スカラプロセッサは，浮動小数点条件コードを1組しか持たないことから，ソフトウェアパイプライニングによる高速化が難しい．このため，OPIが1.35，CPOが1.63と良い値ではない．また，固定小数点演算により絶対値演算を行っていることから，etc.が47.8%と高い．

3.3.9 mdljsp2

mdljdp2が倍精度演算であるのに対し，本プログラムは単精度演算である．VPP500スカラプロセッサは，単精度/倍精度浮動小数点に関する，ロード，演算，ストアの素性能が同じであることから，mdljdp2のほうがオペランド長が大きく，キャッシュヒット率OP$が94.95%と低いことを除けば，酷似した測定結果となった．

3.3.10 swm256

DOループのループアンローリングおよびソフトウェアパイプライニングが可能である．各インタロックが小さく，alvinnに似た特性を示している．ただし，alvinnに対してfaddが32.5%，loadが28.3%と，演算が多くロードが少ない．このため，OP$が96.27%と同程度であるにも関わらず，Wが5.3%と低い．その他のインタロックも低く，OPIが1.73，CPOが0.81，FORが33.19%と最も良い値を示し，正規化後の対VP2600性能比でも2.80と良い値を示している．

3.3.11 su2cor

DOループのループアンローリングおよびソフトウェアパイプライニングが可能である．tomcatvと同様，ロードが連続して発行される一方，キャッシュヒット率OP$が78.62%ときわめて低いため，Wが22.3%と高い値を示している．このためCPIが2.45と悪く，正規化後の対VP2600性能比も1.51と良くない．

3.3.12 hydro2d

DOループのループアンローリングが可能である．fcmpが7.3%と浮動小数点比較が多い．VPP500スカラプロセッサは，浮動小数点条件コードを1組しか持たないことから，浮動小数点比較を含めたソフトウェアパイプライニングによる高速化が難しい．このため，D.CCが29.5%と，mdljdp2およびmdljsp2に似た特性を示している．

3.3.13 nasa7

DOループのループアンローリングおよびソフトウェアパイプライニングが可能である．tomcatvやsu2corと同様，ロードが連続して発行される一方，キャッシュヒット率OP$が77.34%ときわめて低いため，Wが40.5%ときわめて高い値を示している．このため，CPIが3.33と最も悪く，正規化後の対VP2600性能比も1.56と良くない．

3.3.14 fpppp

ループアンローリングの困難な，浮動小数点演算のランダムな集まりである．最内ループに条件分岐が少なく，かつ，ステートメント量がきわめて多い．最内ループの命令サイズが約64Kバイトもあり，IF$が94.21%，D.IFが25.1%であることが示すように，命令キャッシュのヒット率が低い．これは，doducにも見られる特性である．典型的なステートメントおよび対応する命令列を示す．

```
A=A+(B00*C00+B01*C01+B02*C02+B03*C03
    +B04*C04+B05*C05+B06*C06+B07*C07
```

```
                (途中省略)
      +B24*C24+B25*C25+B26*C26)*D
-----------------------------------------
 1: f.load B00 ---+
 2: f.load C00 ---+
 3: f.load B01    +
 4: f.load C01    V
 5: f.load B02  f.mult B00*C00 ---+
 6: f.load C02                    +
 7: f.load B03  f.mult B01*C01 ---+
 8: f.load C03                    +
 9: f.load B04  f.mult B02*C02    +
10: f.load C04                    +
11: f.load B05  f.mult B03*C03    V
12: f.load C05  f.add  B00*C00+B01*C01
13: f.load B06  f.mult B04*C04
14: f.load C06
15: f.load B07  f.mult B05*C05
16: f.load C07  f.add B02*C02+B03*C03
                (途中省略)
*1: f.mult *D
*2: f.add  +A
    -E.FP-      .. 演算結果待ちインタロック
*3: f.store A
```

離散的な変数 B および C をロードするために，レジスタおよび 12 ビット以上のインデックス値によるアドレス指定が必要である．2.2.2節に述べたように，このようなロード操作は 40 ビット長操作にしかなく，1 命令に最大 2 個までしか操作を入れることができないため，OPI が 1.25 と低い．また，最終的に A をストアする際に，FQ 内の演算終了を待つために，E.FP が 29.0%と非常に高い値を示している．

3.4 考察

測定結果から特に注目すべきインタロックを取り出し，表 3.5にグラフとして示す．この結果から，VPP500 スカラプロセッサに関して，以下のことが言える．

- W が高い値を示す，tomcatv，su2cor，nasa7 では，オペランドキャッシュが 32K バイトかつダイレクトマップ方式であることから，64K バイトかつ 16 ウェイの VP2600 スカラプロセッサに比べるとキャッシュヒット率が低く，折角ソフトウェアパイプライニングを適用しても，これに見合うだけの十分な性能が出ない．キャッシュ容量およびウェイ数の増加による大幅な性能向上が期待できる．

- E.FP が高い値を示す，wave5，ora では，FQ が良く機能しており，依存関係のある浮動小数点演算が多数 FQ にキューイングされている．しかし，演算結果を得てストアする際のインタロックが大きく，加減乗算に要するサイクル数 4 そのものを削減する努力が必要である．

- D.IF が高い値を示す，doduc，fpppp では，命令キャッシュのヒット率が低い．特に fpppp では，基本ブロックに属する命令が約 64K バイトと巨大であることから，ウェイ数を増やすことよりも，容量を増やすことが性能向上のために有効である．

- D.CC が高い値を示す，mdljdp2，ear，mdljsp2，hydro2d では，浮動小数点比較に 4 サイクルを要すること，または，条件コードレジスタが 1 組しかないことが性能向上の妨げになっている．比較に要するサイクル数を削減することにより，この 4 本のプログラムに対して効果がある．また，サイクル数を削減できなくても，複数個の条件コードレジスタを設けることにより，比較を含めたループアンローリングおよびソフトウェアパイプライニングの適用が可能となり，ear および hydro2d に対して効果がある．

- alvinn，swm256 のように，キャッシュヒット率が高く，かつソフトウェアパイプライニングの適用により OPI を高くできる場合には，本長形式命令語方式がきわめて有効に動作し，高い性能を引き出せる（SPECratio は各々，307.6，165.5[11]）．

さて本アーキテクチャは，SPECfp92 が登場する以前に，livermore ループなどの比較的小さなベンチマークプログラムを参考にしながら，限られたハードウェア資源の中でいかに高性能を引き出すかの工夫を施し，設計したものである．alvinn，swm256 において高い性能を発揮できたことは，当初の設計通りである．一方，SPECfp92 を走行した結果，キャッシュ容量，ウェイ数，浮動小数点演算レイテンシ，浮動小数点条件コードレジスタ数に関して改善すべき点が明らかになった．

表 3.5: 顕著なインタロック

	W	E.FP	D.IF	D.CC
tomcatv	xx	xx		x
su2cor	xxxx	xx		
nasa7	xxxxxxxx	xx		
wave5		xxxx		x
ora		xxxx		x
doduc		xxx	xxx	xx
fpppp		xxxxx	xxxxx	
mdljdp2				xxxxxxx
mdljsp2				xxxxxxx
ear		x	xx	xx
hydro2d	x	xx	x	xxxxx
alvinn	xxxx			
swm256	x	x		

x は 5%を表す.

キャッシュ容量，ウェイ数，浮動小数点演算レイテンシについては，ハードウェア物量を鑑み，当時，最大限努力して実現したものである．しかし，浮動小数点条件コードレジスタが足りないために，ループアンローリングおよびソフトウェアパイプライニングが適用できず，本長形式命令語方式が有効に機能しなかったことは，大いに反省しなければならない．

3.5 今後の展望

本節では，今後，VPP500スカラプロセッサの開発時よりも，より多くのハードウェア資源を活用できる場合を想定し，当時は達成できなかった，アーキテクチャ上およびインプリメント上のいくつかの点について，他の研究成果に触れながら考察を行い，さらに高性能を追求するための指針を示す．

3.5.1 ソフトウェア・パイプライニングの追加アシスト機構

ソフトウェア・パイプライニングについては，現在，ハードウェアアシスト機構を含めて多くの研究が行われている [23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 41, 42, 43]．提案されている主なアシスト機構は，(1) ローテーティング・レジスタファイル；(2) プレディケーション機構（条件つき命令実行機構）；である．

ローテーティング・レジスタファイルは，たとえば，モジュロ・スケジューリングにより，N 回目のループを実行する命令列と，N+1 回目のループを実行する命令列とが混在する場合に，N 回目で使用する変数と，N+1 回目で使用する変数とを効率良く移しかえる機構である．VPP500 スカラプロセッサの場合，汎用レジスタおよび浮動小数点レジスタを各 32 本装備しており，十分な数のレジスタがある．また，長形式命令語中の空き領域を使って変数を移しかえることができるため，特別な機構を装備しなくても，効率の良いモジュロ・スケジューリングを行うことができる．

一方，プレディケーション機構は，条件分岐命令を使わずに，条件文を実行する機構である．VPP500 スカラプロセッサでは，少段数パイプラインと PC 相対分岐命令を採用することにより，条件分岐命令の実行そのものを高速化し，条件成立時にも不成立時にも分岐ペナルティをほぼ 0 とすることができた．しかし，今後，パイプライン段数を細かく刻むことにより，load-use ペナルティの削減や，クロックサイクルの高速化などを図る場合には，VPP500 スカラプロセッサと同様の高速な条件分岐機構を踏襲していくことが困難になる．このような場合には，プレディケーション機構を装備することにより，条件分岐命令そのものを削減し，高速化を図る必要がある．VPP シリーズの後継機種では，実際に，条件コードレジスタ数の増加に加え，プレディケーション機構の搭載を予定している．

なお，プレディケーション機構には，おおまかに，完全プレディケーション（Full Predication）と部分プレディケーション（Partial Predication）とがある．完全プレディケーションのほうが効果が高いことは言うまでもない．しかし，ハードウェアのコストを下げるために，条件コードの値に応じてレジスタ間転送を行うか否かを決定する「条件つき移動命令」（Conditional Move）や，条件コードの値に応じてレジスタ A，B のいずれかをレジスタ C に転送する「選択命令」（Select 命令）だけをインプリメントすることも考えられる．部分プレディケーションと呼ばれるこれらの機構を採用した場合でも，ある程度の効果が得られることが報告されている [44]．

3.5.2 投機的実行のためのアーキテクチャサポート

次に，投機的実行のためのアーキテクチャサポートについて考える．最も重要な課題は，「いかに，投機的実行にともなう好ましくない副作用をなくし，自由な投機的実行を可能とするか」である [36, 37, 38, 39, 40]．

たとえば，キャッシュミスや load-use ペナルティの影響を小さくするためには，ロード命令をできるだけ早期に発行することがきわめて有効である．ところが，条件分岐命令を越えて，上方にロード命令を移動した結果，アドレスの有効性が保証できない場合があり，起こるはずのない記憶保護例外など，好ましくない副作用が発生することがある．一般には，このような副作用はプログラムの異常終了を引き起こすため，条件分岐命令を越えてロード命令を移動することは，かなり危険である．

しかし，命令スケジューリング手法の中でも，ロード命令の先行実行は最も効果のある方法であるため，不正なアドレスに起因する好ましくない例外をいかに抑えるかは，非常に重要な課題である．現在考案されている主な機構は，(1) 副作用を引き起こす命令にあらかじめマークを付ける方法；(2) 例外を報告しない命令を新たに設ける方法；である．

(1) は，条件分岐を越えて移動したロード命令に，「副作用が起こり得る」ことを示すマーク，および，ハードウェアが副作用を握りつぶすか否かを決定する分岐条件を付加する方法である．記憶保護例外などの副作用は，ソフトウェアに例外を報告する前に，一旦ハードウェア内部に保留しておき，分岐条件が確定したあとに，実際に例外を報告する．ただし，この方法では，複数の条件分岐を越えてロード命令を移動するためには，複雑な分岐条件をロード命令に付加しなければならないことから，現実的には，せいぜい 1 つの条件分岐を越えられるにすぎない．

一方，(2) は，投機的実行のために，専用のロード命令を設ける方法である．このロード命令は，例外を検出した場合には，ロードデータの格納先レジスタに印をつけることのみを行い，例外そのものは報告しない．レジスタに付いた印は，そのレジスタの内容を使用する全ての演算の格納先レジスタに伝搬し，印の付いたレジスタの内容を主記憶にストアする時に，はじめて例外が報告される．この方法によれば，複数の条件分岐命令を越えて，ロード命令を移動することができる．さらに，ストア命令だけは，正しい条件分岐を経由した後に実行するようスケジューリングを行うことにより，ロード結果を使用する演算命令についても，条件分岐を越えて自由に移動することが可能となる．

VPP500 スカラプロセッサの次に開発した VPP300 スカラプロセッサからは，実際に (2) の方法を採用しており，効果を期待している．

3.5.3 分岐予測の確度向上による命令フェッチの高速化

前節において述べたように，今後，高速化のために，パイプライン段数を細かく刻む方向に進む場合には，条件分岐命令の高速化手法として，VPP500スカラプロセッサにおいて採用した方法を補完する，分岐予測機構を装備することが必要となってくる．

分岐予測機構には，(1) あらかじめ分岐命令ごとに分岐予測方向が決定されている静的分岐予測；(2) プログラムの実行時にハードウェアが分岐方向を予測する動的分岐予測；がある．

静的分岐予測

まだ，投機的実行を採用しておらず，分岐予測が外れても，それほどのペナルティとならなかった時代のプロセッサの多くは，ハードウェアを軽量化できる静的分岐予測を採用している．

たとえば，MIPS や SuperSPARC のインプリメントは，条件分岐方向を全て分岐側に予測する静的分岐予測を採用している [45]．一方，MC88000 は，全て非分岐側に予測する [46]．また，PA7000 は，ループ構造に代表される後方分岐の場合は分岐側，前方分岐の場合は非分岐側を予測する方法を採用している．

コンパイラが分岐方向を予測し，ハードウェアに対していずれの方向を優先すべきかのヒントを与える方法も，静的分岐予測に分類することができる．この方法は，PowerPC や PA-8500[60] が採用している．

一方，このようなハードウェアの静的分岐予測が当たる確率を高めるために，コンパイラが条件分岐の方向を予測し，条件分岐が起こることなく，できるだけ命令語がストレートに流れるようにすることにより，高速化を図る研究結果も報告されている [57, 59, 61]．

VPP300 スカラプロセッサからは，コンパイラが分岐命令中の 1 ビットに分岐予測情報を与える静的分岐予測を採用しており，効果を期待している．

動的分岐予測

プロセッサが，より多くの命令を同時に実行するために，条件分岐命令を越えた投機的実行を行うようになると，分岐予測の確度を上げることがきわめて重要になってくる．このため，最近では，動的分岐予測を行うプロセッサが登場するようになってきた．

たとえば，DEC 社の 21064 は，命令キャッシュ中の各命令ごとに，前回の分岐方向を記憶する 1 ビットの動的分岐予測情報を保持している．

また，さらに確度を上げるために，より多くの履歴情報を利用して分岐予測を行う研究が数多く行われており [47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 58]，現在では，以下のような動的分岐予測手法が考案されている．

2 ビットのアップ/ダウンカウンタを一つだけ用意する方法:

全ての条件分岐命令に対して，ただ一つのカウンタを使用する方法である．たとえば，カウンタの値が 0 または 1 の場合は分岐なしを予測し，2 または 3 の時は分岐ありを予測する．実際に分岐しなかった場合にはカウンタの値を 1 だけ減じ，分岐した場合にはカウンタの値を 1 だけ増加する．すなわち，2 回以上連続して分岐した場合，次も分岐すると考え，2 回以上連続して分岐しなかった場合には，次も分岐しないと考える方法である．

過去の分岐パターンに関連づけた 1 レベルの分岐予測:

最近の分岐/非分岐の履歴を N ビットのシフトレジスタとして記憶した値を用いて，2^N 個からなる 2 ビットのカウンタ群から一つを選択する方法である．次の分岐命令に到達する前に，分岐方向を予測することができる特徴がある．

分岐命令のアドレスに関連づけた 1 レベルの分岐予測:

分岐命令のアドレスに応じて，2^N 個からなる 2 ビットのカウンタ群から一つを選択する方法である．カウンタの選択方法として，(1) 分岐命令のアドレスの下位 N ビットを用いる方法；(2) 分岐命令のアドレスの下位 N ビットに，最近の分岐/非分岐の履歴を N ビットのシフトレジスタとして記憶した値を排他論理和したものを用いる方法；が考案されている．後者の方法は，同じ分岐命令であっても，それまでの分岐/非分岐の履歴により，予測される分岐方向が異なることを考慮したものである．

過去の分岐パターンおよびアドレスに関連づけた 2 レベルの分岐予測:

過去の分岐パターンと分岐命令のアドレスの両方を用いて，2 次元に配列された $2^M \times 2^N$ 個のカウンタ群から一つを選択する方法である．分岐/非分岐の履歴を表す M ビット，および，分岐命令のアドレスの下位 N ビットを用いる．

SPECint92[15, 16, 17] のように，ユーザモードが大部分を占めるプログラムの場合，1 レベル予測よりも 2 レベル予測のほうが効果が高く，ハードウェアの複雑度に応じて効果が上がる．しかし，カーネルモードが 5%以上を占めるプログラムの場合には，エイリアシング，すなわち，異なる分岐命令であるにもかかわらず，アドレスの下位 N ビットが同じ値になることにより，分岐予測が混乱するという現象が多発するために，2 レベル予測よりも 1 レベル予測のほうが性能が良いケースがあることが報告されている [52, 53]．これは，同量の 2 ビットカウンタを用いて比較した場合，2 レベルの分岐予測よりも 1 レベルの分岐予測のほうがエイリアシング問題が起きにくいことに起因している [56]．

一方，最近発表された，HP 社の PA-8500[60] は，3.5.3節において述べた静的分岐予測手法と動的分岐予測手法とを効果的に組み合わせている．これは，2 ビットのアップダウンカウンタを「分岐方向により加減する」のではなく，「静的分岐予測の当たり外れにより加減する」も

のである．静的分岐予測が外れる傾向を示す分岐命令に対しては，動的分岐予測が適用される．この手法は，コンパイラが分岐方向を予測できるものについては，確実にその予測に従い，予測が困難な分岐については，ハードウェアが履歴情報を用いて予測を行うという，きわめて理にかなったものであると言える．

3.5.4 キャッシュ性能の改善による命令フェッチの高速化

序論において述べたように，論理素子に対する記憶素子の相対的な動作速度は，次第に低下する傾向にある．このため，最近では，命令キャッシュに対して，いかに迅速に主記憶上の命令列を供給するかに関して，研究が行われている [62, 63].

比較的簡単なハードウェア機構により実現できる高速化手法として，ハードウェアが，連続する複数のキャッシュラインに対してプリフェッチを行う方法（Next-line Prefetching）がある．ラインサイズを大きくするよりも，より効果が高いことが報告されている [62].

VPP500では，次の命令キャッシュラインのプリフェッチは行っていない．このような方法を採用することにより，高速化が期待できると考えている．

さらに，分岐予測手法と関連した研究 [63] では，分岐先アドレスの命令をプリフェッチする際の方針として，(1) 常に正しい分岐方向側をプリフェッチする理想的なモデル；(2) 分岐予測が適度に当たり，予測が外れたことによる無駄なプリフェッチがまれに発生するモデル；(3) ノンブロッキングキャッシュと (2) を組み合わせたモデル；(4) 分岐方向が確定するまでプリフェッチしないモデル；を仮定し，Next-line Prefetching と組み合わせた性能比較を行っている．この結果，命令キャッシュミスのレイテンシが小さい場合には，Next-line Prefetching を行う (3) が最も効果が高く，一方，レイテンシが大きい場合には，Next-line Prefetching を行わない (3) または (4) が効果が高いことが報告されている．

3.5.5 参照の局所性/規則性を利用したオペランドフェッチの高速化

分岐予測機構と同様に，参照の局所性や規則性を利用することにより，オペランドフェッチについても高速化しようという研究が行われている．オペランドフェッチの高速化には，大きく，(1) ハードウェアによるプリフェッチ；(2) ソフトウェアによるプリフェッチ；とがある．

ハードウェアによるプリフェッチ

ロード命令におけるアドレス指定は，BASE+OFFSET であることが多い．もし，実際に BASE+OFFSET を計算する前に，アドレスを予測し，キャッシュへのアクセス要求を発行することができれば，load-use ペナルティを削減することができる．この考えに基づき，BASE と OFFSET の論理和を用いて，キャッシュへのアクセス要求を早期に発行する研究が報告されている [64]．コンパイラができるだけ小さい OFFSET を使用することにより，BASE と OFFSET の論理和が，実際の BASE+OFFSET の値が該当するキャッシュのラインにヒットする確率を高めることができる．

プリフェッチエンジンを設ける方法も提案されている [65, 66]．プリフェッチを起動する命令アドレスとプリフェッチパターン（先頭アドレス+ディスタンス等）を登録しておき，条件が満たされた場合に，ハードウェアが自動的にプリフェッチを起動する方法である．具体的には以下の機構が提案されている．

Basic Reference Prediction:

ロード命令の命令アドレスごとに，(1) 最後に使用したオペランドアドレス；(2) 最後に使用したオペランドアドレスの差分；(3) 分岐予測に用いたものと同様の 2 ビットカウンタ；を設け，次にロード命令を実行する際のオペランドアドレスを予測し，プリフェッチを行う方法である．ただし，プリフェッチを行うタイミングは難しく，早過ぎる場合には使用中のキャッシュ内容を追い出す結果となり，遅過ぎる場合には効果が薄い．

Lookahead Reference Prediction:

Lookahead-PC（仮想的なプログラムカウンタ）を使って，オペランドが必要な時期にちょうど間に合うよう，プリフェッチを行う方法である．Lookahead-PC には，3.5.3節において述べた動的分岐予測機構を用い，常に，オペランドフェッチのレイテンシ分だけ先の PC を予測する．この Lookahead-PC の値がプリフェッチエンジンに登録されているロード命令の命令アドレスに一致した時，プリフェッチを起動する．規則的なループには効果があるものの，小さいループや，三角行列を扱うような不規則なループでは，効果が上がらない．

Correlated Reference Prediction:

さらに，3.5.3節において述べた，分岐履歴を保持するシフトレジスタの値をも動員する方法である．分岐履歴の値に応じて，複数のプリフェッチパターン（先頭アドレス+ディスタンス）の中から，一つを選択することにより，三角行列を扱うループのような，より複雑なループにも対応することができると考えられている．

ソフトウェアによるプリフェッチ

ハードウェアを複雑化することなく，ソフトウェアがロード命令を先行して発行することにより，高速化を図る手法である [67, 68, 69, 70, 71].

非数値計算プログラムでは，リストアクセス（load-load-use），および，キャッシュブロックを越えるような距離の大きいストライドアクセスの出現頻度が高く，このようなロード命令を

先行実行することにより，キャッシュミスを含むload-useペナルティを効果的に削減することができる[70].

一方，数値計算プログラムでは，ループ構造を利用して，効果的なプリフェッチを行うことができる．リストアクセス，および，より小さな距離を含むストライドアクセスについて，ロード命令をスケジューリングすることにより，非数値計算プログラムよりも大きな効果を得られることが報告されている[71].

VPP500では，すでに，2.2.4において述べた，ソフトウェアによるプリフェッチのためのアシスト機構を採用している．上記のような最適化手法をきめ細かく適用することにより，さらに性能を向上できると期待している．

3.5.6 レジスタの分割使用による並列度の向上

最近では，64ビット長またはそれ以上の演算レジスタを2〜8分割し，2〜8個の独立した演算を同時に行うことにより高性能を図った，商用プロセッサが登場している[73, 74, 75]．これらは，3次元グラフィクスや画像データの圧縮/展開など，いわゆるマルチメディア処理を高速化するために登場した機構である．

このような機構は，今後，さらにレジスタ数を増加し，演算の並列度を上げることによる高速化の方向に進み，しだいに，ベクトル演算機構に近づいていくものと考えている．

一方，VPP500では，2.2.4節において述べたように，スカラプロセッサと緊密に連係するベクトル演算機構を付加しており，非同期命令として高速にベクトル演算を実行することができる．スカラプロセッサは，むしろ，ベクトル化の困難な処理を高速化する方向に進み，ベクトル化が可能な部分については，外部の機構に任せることが，全体のコストパフォーマンスの向上のために適切であると考えている．

3.5.7 長形式命令語方式におけるロードモジュール互換技術

スーパスカラ方式では，ロードモジュールの非互換が問題になることはない．一方，長形式命令語方式では，一般に，異なるモデル間においてロードモジュール互換を維持するために，特別の考慮を必要とする．具体的には，(1) 命令のレイテンシの違いによる非互換；(2) 並列度の違いによる非互換；をいかに吸収するかが大きな問題となる．このような問題を解決する方法として，以下の方法が考えられる．

全モデルのロードモジュールを保持:

各モデルのロードモジュールを全て保持しておく方法である．Multiflow社のTRACE-300シリーズが採用している[22]．ただし，この方法は，再コンパイルを前提としているため，あとから新しいモデルを開発する場合に備えて，ロードモジュールと同様にソースプログラムも厳重に管理しなければならない欠点がある．

ハードウェアによる命令変換:

全てのモデルの命令列を実行できるよう，デコーダや機能ユニットを冗長に装備したハードウェアを構成する方法である．異なるモデル用のプログラムであっても，低能低下が許されない場合には，ハードウェアコストが増加するとしても，この方法を採用する必要がある．

ソフトウェアによる命令変換:

次章においても述べるように，ソフトウェアが，実行前に命令列を変換する方法である．命令語のフェッチ時に発生する，ページフォルトを契機として，オペレーティングシステムが，現在のモデルで動作するよう，命令列を変換する方法が提案されている[72].

2.2.4節において述べたように，VPP500では，異なる命令語の間にデータ依存関係が存在する場合には，ハードウェアが命令の実行順序を保証しており，(1) 命令のレイテンシの違いによる非互換；が生じない優れたアーキテクチャとなっている．

一方，将来，VPP500が採用している64ビット長の命令語を拡張することになった場合，(2) 並列度の違いによる非互換；が問題となることが予想される．この場合には，ハードウェアコストを考慮しながら，「ハードウェアによる命令変換」または「ソフトウェアによる命令変換」を行うことになると考えられる．

3.6 おわりに

本章では，VPP500スカラプロセッサの性能測定項目について述べ，SPECfp92を用いた測定結果に基づき，VP2600スカラプロセッサと比較しながら考察を行った．

VPP500スカラプロセッサの開発に際しては，限られたハードウェア資源の中でいかに高性能を引き出すかの工夫を施し，VP2600よりも遅いサイクルタイムであるにも関わらず，同等の性能を達成した．

一方で，さらに高性能を達成するためには，浮動小数点条件コードレジスタ数の増加が必要であること，また，直接ハードウェア量の増加につながるけれども，キャッシュ容量，ウェイ数，浮動小数点演算に要するサイクル数について改善することにより，さらに高性能を引き出せることが明らかになった．

最後に，今後，より多くのハードウェア資源を活用できる場合を想定し，アーキテクチャ上およびインプリメント上のいくつかの点について，さらに高性能を追求するための，以下のような指針を示した．

- プレディケーション機構の装備による，条件分岐命令そのものの削減．
- 投機的実行のための，例外を検出しないロード命令の新設．
- 静的および動的分岐予測機構を装備することによる，分岐ペナルティの削減．
- 連続する複数の命令キャッシュラインに対するプリフェッチ機構．

これらは，実際に，VPPシリーズの後継機種を開発する際の指針となっており，効果を期待している．

Chapter 4

命令エミュレーションに基づくMアーキテクチャの構成方式と性能評価

我々は，Mアーキテクチャ上で動作するオペレーティング・システムを動的命令変換手法によりSPARCシステム上で動作させる実験システムを開発した．動的命令変換手法は，M命令をSPARC命令に変換および蓄積しながら実行する命令エミュレーション手法であり，変換後SPARC命令を再利用する頻度が高いほど，高性能を得ることができる．本章では，動的命令変換手法の動作原理について詳述し，ベンチマーク・プログラムを走行させて得た結果に基づき，詳細な性能評価を行う．実運用状態に近いI/O頻度を有し約40%をスーパバイザモードで走行するジョブについても，変換後命令の再利用率は99.98%に達し，オペレーティング・システムを含めてエミュレーションを行った場合でも，動的命令変換手法がきわめて有効であることを明らかにする．一方，M命令のワーキングセットの大きさに依存してSPARCの命令キャッシュヒット率が敏感に変化すること，また，このために，変換後SPARC命令の実行に際して，命令キャッシュヒット率を向上させるための工夫がきわめて重要であることを明らかにする．

4.1 はじめに

Mアーキテクチャ[76]は，汎用計算機として長い歴史を有するアーキテクチャである．このため，これまで蓄積されてきたソフトウェア資産は膨大なものとなっており，今後もMアーキテクチャに基づくハードウェアを開発し維持していく必要がある．一方，比較的小規模な計算機システムとして，UNIXやWINDOWS-NTといった低価格なシステムが次第に普及し，Mアーキテクチャのソフトウェア資産にこだわる必要がない場合には，アーキテクチャの選択肢が広がりつつある．

このような状況の中で，Mシリーズ計算機システムのうち，特に小規模なシステムは，UNIX

やWINDOWS-NTシステムとコストが競合する位置にあるため，いかにコストを抑えるかが非常に重要な問題となっている．この問題を解決する手段の一つとして，量産効果によりコストの小さい，UNIXやWINDOWS-NTシステムのハードウェアを利用し，エミュレーション技術によりMアーキテクチャを実現する方法が考えられる．

ある命令セットアーキテクチャに基づいて作成されたプログラムを，異なる命令セットアーキテクチャに基づくハードウェアシステム上で動作させる手法は，マイクロプログラムによるエミュレーション技術として，以前から広く知られている [77, 78, 79, 80, 81, 82].

しかし，その後，マイクロプログラム（ファームウェア）の入れ換えを基本とするエミュレーション技術は，ほとんど使われなくなり，エミュレーション技術は過去のものになったかに見えた．

一方，ここ十数年の間，半導体テクノロジの進歩，および，プロセッサ・アーキテクチャの進歩により，マイクロプロセッサの性能は年率25%の向上を見せており，今後も継続する勢いである．特にプロセッサ・アーキテクチャに関しては，(1) レジスタ-主記憶間演算命令を柱とするCISC型命令セットが，レジスタ間演算命令を柱とするRISC型命令セットに置き換わるといった，命令セットアーキテクチャの変化；(2) ハードウェアに課せられていた主記憶参照順序の保証 [76] やプリサイス・インタラプト [76, 14] などの制約事項が，Relaxed-Memory-OrderやDefferred-Trap[86] の導入により緩和されるといった，プログラム実行モデルの変化；などが，ハードウェアの軽量化および高速化を助けてきた．

最近では，このように，プロセッサの命令セット・アーキテクチャやプログラム実行モデルが変化していく中で，従来のソフトウェア資産を動作させるために，再び，エミュレーション技術が顧みられようとしている．現在，主流となっているマイクロプロセッサでは，マイクロプログラム（ファームウェア）の入れ換えによるエミュレーションは困難であり，厳密にはプログラム（ソフトウェア）によるシミュレーションと呼ぶ手法を使用する（本章では，これもエミュレーションと呼ぶことにする）．

従来のソフトウェア資産を移植する方法としては，資産の形態に応じて，おおまかに，以下のような手法が考えられている．

再コンパイル手法:

高級言語により記述されたソースプログラムが存在する場合は，主に再コンパイルにより，新しいプロセッサ上で動作するロードモジュールを生成することができる．

静的命令変換手法:

アセンブリ言語により記述されたソースプログラムが存在する場合，または，ロードモジュールしか存在しなくても，命令とデータが区別でき，たとえば分岐先アドレスが相対アドレス指定であるために，分岐先アドレスが比較的容易に特定できる場合は，あらかじめロードモジュール全体を新しいプロセッサの命令セットに変換することができる．一般に，アプリケーション・プログラムは，人手の介入無しに変換できることが多い．しかし，オペレーティング・システムのように，ハードウェアと密接に連係するプログラムの場合，命令セットとしては現れないプログラムの意図を含めて変換する必要があり，きわめて多くの人手の介入を必要とする．

動的命令変換手法:

ロードモジュールしか存在せず，命令とデータの区別が不明確である，または，たとえば分岐先アドレスがレジスタ間接指定であるために，実行してみないと分岐先アドレスが容易に特定できない場合は，ロードモジュール全体をあらかじめ変換することは困難である．この場合，実行時に必要に応じて新しいプロセッサの命令セットに変換し実行する，動的命令変換手法を採用することになる．

インタプリタ手法:

動的命令変換手法と同様，ロードモジュールしか存在しない場合に，命令を逐次解釈実行する手法である．命令変換によるプログラムサイズの巨大化という問題が生じないため，記憶容量を小さくできる利点がある．しかし，一般に実行速度は最も低速である．

再コンパイル手法を採用することができれば，プロセッサ本来の性能を引き出し，最高の性能を得ることができる．ところが，ソフトウェア資産を開発者以外がソースプログラムとして利用できることは稀である．多くの場合，ロードモジュールしか利用できないため，静的命令変換手法，動的命令変換手法，または，インタプリタ手法が重要となる．

最近では，単なるプロセッサ・アーキテクチャの世代交代のためではなく，他のアーキテクチャ上で動作する優れたソフトウェアを搭載するために，ロードモジュールの変換および実行を行う商用システムが登場している．DEC社のFreePort Expressは，静的命令変換手法を採用しており，SunOS上で動作するアプリケーション・プログラムおよびライブラリから，Digital UNIX上で動作するプログラムを生成する [83]．またSUN社のWabiは，動的命令変換手法およびインタプリタ手法を採用し，Windowsアプリケーション・プログラムをを SunOS上で動作可能としている [84].

さて，以上に挙げた手法は，実際のシステムではアプリケーションプログラムにのみ適用されている．オペレーティングシステムについては，移植先のオペレーティングシステムの機能を利用しているか，または，移植先のハードウェア専用に新規に開発している．ところが，Mアーキテクチャ上で動作しているオペレーティングシステムの機能は，ファイルシステムや文字コードの非互換に代表されるように，UNIXやWINDOWS-NTシステムでは肩代りすることができない．また，規模が大きく，かつ，アセンブリ言語により記述されている部分が非常

に多いため，新しいハードウェア専用に新規に開発することはきわめて困難である．このため，Mアーキテクチャ上で動作するアプリケーションプログラムをUNIXやWINDOWS-NTシステムのハードウェア上で動作させるためには，オペレーティングシステムを含めて動作させなければならない．特に，オペレーティングシステムを動作させるためには，Mアーキテクチャが規定するアドレス変換機構やI/O割り込み検出機構といった，従来のエミュレーションシステムでは必要のなかった機能をエミュレートしなければならず，これらをいかに効率よく実現するかが大きな問題となる．

このような動機づけから，我々は，命令エミュレーション手法が，Mアーキテクチャ上で動作するオペレーティング・システムおよびユーザ・プログラムに対して，どの程度有効であるか，また，どのような問題点が存在するかを調査することを目的として，Mアーキテクチャ上で動作するオペレーティング・システムをアーキテクチャの異なるシステム上で動作させる実験システムを開発した．コストを下げることが第一の目的であるため，新規のハードウェア開発は行わない．すなわち，UNIXやWINDOWS-NTシステムのハードウェアに対して，エミュレーションをアシストする機構を加えることは研究対象とせず，ソフトウェアだけでどこまで高速なエミュレーションが可能であるかを追求することにした．

また，我々は，以下に挙げる理由から，エミュレーション方式として動的命令変換手法を採用した．

- オペレーティング・システム（以下OSと略する）を静的変換手法によりSPARC命令に変換するには，前述したように，多大な時間と労力が必要となる．
- Mアーキテクチャでは，分岐先アドレスがレジスタ間接指定であるために，実行してみないと分岐先アドレスが容易に特定できない．
- 可能な限り高速性を追求する．

本章では，4.2節において，Mアーキテクチャに対し動的命令変換手法を適用する際の課題について，また，4.3節において，本実験システムの概要を述べる．続いて，4.4節および4.5節において，動的命令変換手法の動作について，また，4.6節において，自己変更プログラムを正しく動作させるための考慮について説明する．

4.7節では，実際にベンチマーク・プログラムを走行させて得た結果に基づき，詳細な性能評価を行う．特に，M命令のワーキングセットの大きさに依存してSPARCの命令キャッシュヒット率が敏感に変化すること，このために，変換後SPARC命令の実行に際して，命令キャッシュヒット率を向上させるための工夫がきわめて重要であることを明らかにする．

最後に，4.8節において，さらに高速化を図るための方策について検討を加える．

なお，以下では，Mアーキテクチャを M，SPARCアーキテクチャを SPARC と略する．

4.2 動的命令変換手法を適用する際の課題

Mアーキテクチャに対し動的命令変換手法を適用するにあたり，我々は，いくつかの課題に対して，以下のような選択を行った．

【課題 1】

エミュレーションを行うプラットホームとして，どのようなアーキテクチャを選択するかが，エミュレーション性能に大きな影響を及ぼす．変換後命令の大きさを可能な限り小さくし高速化を図るためには，アーキテクチャに対して，(1) Mが規定する16本の32ビット長汎用レジスタ，および，8本の64ビット長浮動小数点レジスタを常駐できるだけのレジスタを保有すること；(2) Big-endian byte orderであること；(3) 固定小数点演算および浮動小数点演算が類似していること；という条件が求められる．また，I/Oを含めたシステムの詳細な仕様が明確であることも重要な条件である．ただし，これらの条件を全て満足することは難しい．特に，精度や例外検出を含めて，Mの浮動小数点演算命令を実行できるアーキテクチャは，M以外に無い．このため，浮動小数点演算は，固定小数点演算の組合せにより実現することにした．その代わり，汎用レジスタの常駐およびBig-endian byte orderに関しては，必須条件とした．

【課題 2】

前節において述べたように，Mのアドレス変換機構をどのように実現するかは，きわめて重要な課題である．変換後命令において明示的にアドレス変換を行うことは可能である．しかし，アドレス変換にはかなりの手間を要するため，高速化を狙う時には不向きである．一方，プラットホームが提供するアドレス変換機構を利用することができれば，アドレス変換を高速に行うことができる．ただし，MのOSはページの管理を4Kバイト単位に行うことから，プラットホームが4Kバイトのページサイズを扱うことができる必要がある．高速性を追求するために，プラットホームの選択範囲が制限されるものの，4Kバイトページを扱えるという条件を必須とした．

【課題 3】

前節において述べたように，変換後命令実行中に，外部割り込みを確実かつ効率よく検出しなければならない．4.4.1および4.4.3節において詳述するように，Mにおいて命令分岐が発生した場合にのみ，割り込み保留状態を検査する方法を採用した．これにより，検査のためのオーバヘッドを最小限に抑えることを狙った．

【課題 4】

動的命令変換が効率良く機能するためには，変換後命令をうまく再利用できること，ま

た，命令を書き換えながら実行する自己変更プログラムにも対応できることが必要である．このための条件として，(1) 分岐命令によりどこから飛び込んできても，アドレスが同じならば同じ変換後命令を利用できること；(2) M命令が書き換えられた場合には，そのM命令に対する変換後命令の範囲が一意に特定でき，その範囲の変換後命令を無効化しても他の命令の実行に影響を及ぼさないこと；が挙げられる．変換後命令の構造および管理方法を決定する際には，この条件を重視した．

【課題5】

Mの4バイトロード命令（L命令）は，オペランドアドレスが4の倍数でなくても，例外を発生することはなく，連続する4バイトをロードする．一方，たとえばSPARCの4バイトロード命令は，オペランドアドレスが4の倍数でない場合，アドレス境界違反例外を発生する．もし，オペランドアドレスの大部分が4バイト境界であるならば，L命令をエミュレートする際に，4バイトロード命令を使用することにより高速化を図ることができる．稀に4バイト境界でない場合には，アドレス境界違反例外を契機として，モニタが1バイトロード命令により再実行すればよい．一方，オペランドアドレスが4バイト境界でない頻度が高い場合，アドレス境界違反例外の発生頻度が高くなるため，常に1バイトロード命令を使用するほうが高い性能が得られる．我々は，2バイト長または4バイト長のオペランドは，大部分が各々のデータ長にアラインされていると仮定し，対応する2または4バイトロード/ストア命令を使用することにした．性能測定時に，ミスアラインに伴うオーバヘッドについて検証する．

【課題6】

Mにおいて規定される各例外の検出をいかにオーバヘッドを少なく実現するかについては，4.4.4節において詳述するように，命令変換および変換後命令実行の中で，なるべく明示的な検査を行わずに検出できるよう工夫を凝らした．

4.3 実験システムの概要

冒頭において述べたようにコストを小さくし，また，前節において述べた課題を解決するために，ハードウェア・プラットホームに求められる条件は，以下の通りである．

- 一般的なUNIXやWINDOWS-NTシステムとして量産されておりコストの小さいシステムであること．
- Mが規定する16本の32ビット長汎用レジスタを常駐させるのに十分なレジスタを有すること．
- Big-endian byte orderであること．
- ページサイズが4Kバイトであること．

以上の条件を満たし，さらに，実験システムを構築するために必要な，I/O機構を含めたハードウェア各部の詳細な資料が容易に得られるという条件を満たすシステムとして，我々は，SuperSPARCを搭載したFujitsu製DS/90システムをプラットホームとして採用することにした．

実験システムの構成を図4.1に示す．DS/90では，SPARCシステム本来のOSは動作させず，ハードウェアに付属するROMモニタ（ハードウェアを初期化しエミュレータを起動するための必要最低限の機能を有するROMプログラム），エミュレーション・モニタ，動的命令変換エミュレータ，および，I/Oエミュレータが動作する．エミュレーション・モニタは，システム全体の制御，Mの制御命令，アドレス変換，および，割り込みのエミュレーションを行う．動的命令変換エミュレータは，浮動小数点演算命令，および，10進演算命令を含む一般命令のエミュレーションを行う．I/Oエミュレータは，DS/90のDISK装置，MT装置，および，RS232C装置を駆動して，OSが認識するDISK装置，MT装置，および，端末装置をエミュレートする．DISK装置にはOSをあらかじめ格納している．エミュレーション・モニタ（以下モニタと略する），動的命令変換エミュレータ（以下動的変換部と略する），および，I/Oエミュレータは，カートリッジMTに格納されており，ROMモニタに対するbootコマンドの投入を契機として，実験システムが起動する．

複数のRS232C装置により，DS/90と接続したSUNワークステーション上では，SunOSの下でF6680端末エミュレータが走行し，各1本のRS232Cを介してI/Oエミュレータと通信することにより，Mの端末画面を提供する．

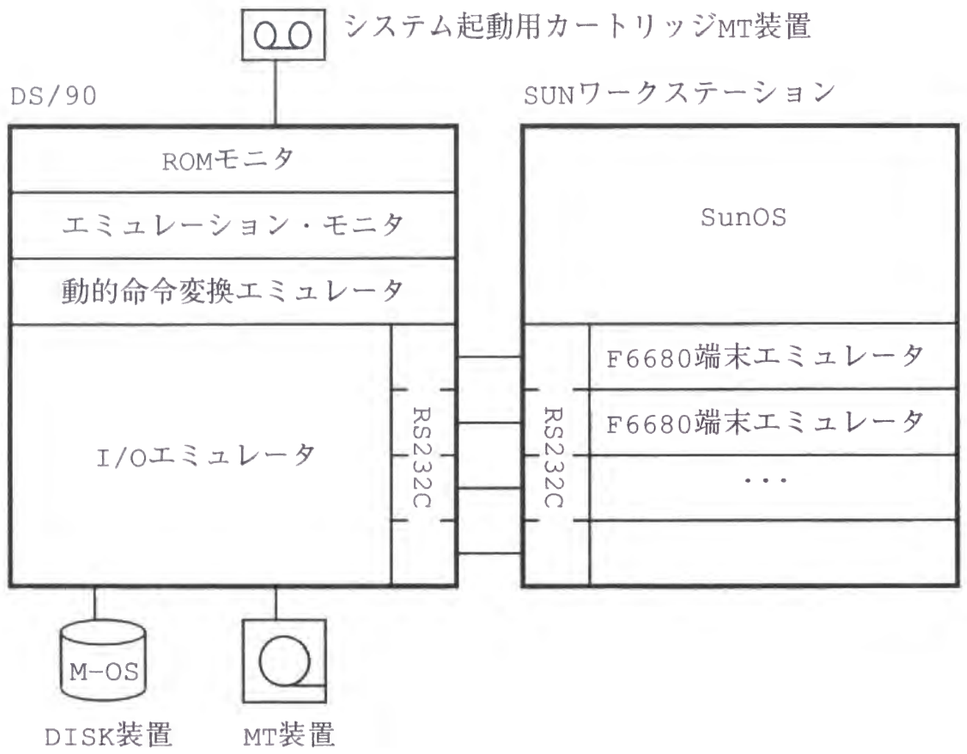


図 4.1: 実験システムの構成

4.4 動的命令変換手法

動的命令変換手法は，大まかに述べると，実行を開始するM命令が，まだSPARC命令に変換されていない場合は，命令変換を行い，変換済みである場合には，変換後のSPARC命令を実行することを繰り返す，命令エミュレーション手法である．変換後のSPARC命令数を極力減らし，高速化を図るために，Mの汎用レジスタの内容をSPARCの汎用レジスタ上に常駐させている．一方，Mの浮動小数点レジスタは，SPARCの主記憶上に割り付けている．これは，Mの浮動小数点形式とSPARCの形式とが異なり，SPARCの浮動小数点演算を直接利用することがないためである．

4.4.1 SPARCレジスタの割り付け

図4.2に，SPARCレジスタの割り付けを示す．%gで表されるグローバルレジスタには，Mの条件コード，プログラムカウンタ，割り込みマスク等の制御データを格納し，その他のレジスタには，Mの汎用レジスタおよび作業領域を割り付けている．なお，汎用レジスタ14だけは，SPARCのフレームポインタとして使用される%fpを避けるために，%g6に割り付けている．詳細は以下の通りである．

CC:

Mの2ビット条件コード．

IA:

Mのプログラムカウンタ．

Address Mask:

Mの有効アドレスをアドレスモードに従い，24ビットまたは31ビットに封じ込めるためのマスク$00ffffff_{(16)}$または$7fffffff_{(16)}$．

OP-base:

SPARCの主記憶に割り付けた，Mの主記憶の先頭アドレス．

IF-base:

アドレス対応表の先頭アドレス（後述）．

FC:

動的変換部が，モニタに対して，例外の通知や制御命令実行の依頼を行う際に使用．

I:

モニタがI/Oからの外部割り込みを認識した際に，本領域に1を設定することにより，動

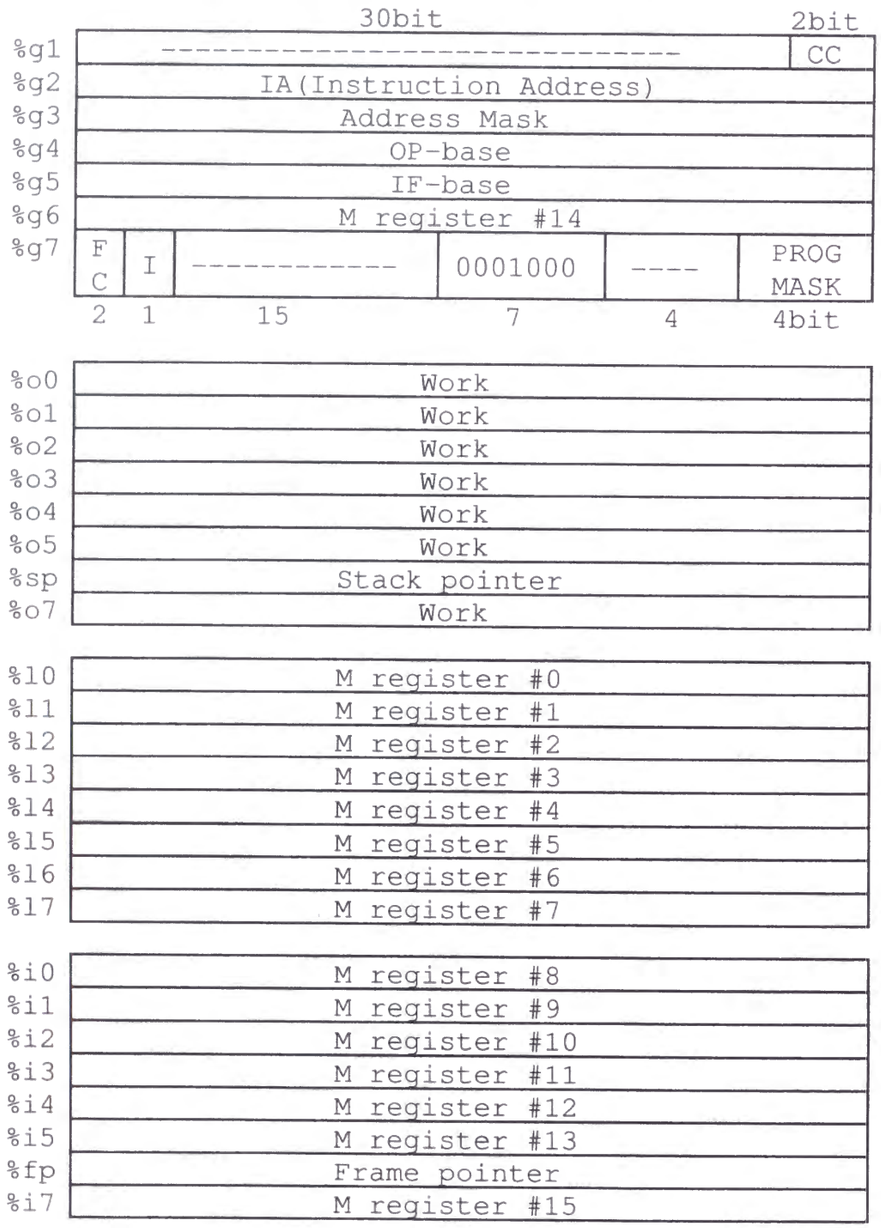


図 4.2: SPARC レジスタの割り付け

的変換部が現在実行中の M 命令列を一旦終結し，モニタへ戻る．

0001000:

動的変換部が条件分岐の高速実行のために使用する定数（後述）．

PROG MASK:

M のプログラムマスク．

Work:

作業用レジスタ．

Stack pointer:

SPARC のスタックポインタとして使用．

Frame pointer:

SPARC のフレームポインタとして使用．

M register #0-#15:

M の汎用レジスタ 0 から 15 として使用．

4.4.2 変換後 SPARC 命令

以上のようなレジスタ割り付けに基づき，動的変換部は，M 命令から SPARC 命令への変換を行う．本節では，図 4.4の太枠中に示した SPARC 命令を例として，変換後 SPARC 命令について説明する．

ロード命令（L）：

$2044_{(16)}$番地のロード命令 “L 2，76（0，13）” は，5 個の SPARC 命令に変換される．最初の 3 命令により，“76（0，13）” の実効アドレス（76+%g0+%i5）を%g3 によりマスクしたアドレスを求め，次の ld 命令により主記憶オペランドを M の汎用レジスタ 2 に対応する%l2 へロードする．最後にプログラムカウンタ%g2 に 4 を加算する．

レジスタ間転送命令（LR）：

$204C_{(16)}$番地のレジスタ間転送命令 “LR 0，2” は，2 個の SPARC 命令に変換される．最初の add 命令により M の汎用レジスタ 2 に対応する%l2 から%l0 へ複写し，次にプログラムカウンタ%g2 に 2 を加算する．

比較命令（CL）：

$204E_{(16)}$番地の比較命令 “CL 0，12（9，12）” は，11 個の SPARC 命令に変換される．

最初の3命令により“12（9，12）”の実効アドレス（12+%i1+%i4）を%g3によりマスクしたアドレスを求め，次のld命令により主記憶オペランドをロードする．subcc命令によりロード結果をMの汎用レジスタ0に対応する%l0と比較し，Mの条件コードを%g1に生成する．最後にプログラムカウンタ%g2に4を加算する．

条件分岐命令（BC）：

$2052_{(16)}$番地の条件分岐命令“BC 13，92（0，15）”は，9個のSPARC命令に変換される．最初のsrl命令およびandcc命令により，%g7中の定数$0001000_{(2)}$をMの条件コード%g1ビットだけシフトし，条件マスク値13と比較する．条件が成立しない場合には，プログラムカウンタ%g2に4を加算して次のM命令すなわちL1:へ分岐する．条件が成立する場合には，分岐先アドレス“92（0，15）”に対応する（92+%g0+%i7）を%g2に格納して，前述したbrk:へ戻る．

浮動小数点演算命令や10進演算命令など，変換後SPARC命令のサイズが大きいものについては，ライブラリを呼び出すSPARC命令列に変換する．

4.4.3 動作原理

動的変換部は，次に示す手順により，M命令からSPARC命令への変換，および，変換後SPARC命令の実行とを交互に行いながら，M命令のエミュレーションを行う．図4.3に初期状態，図4.4に走行状態を各々例示する．

1. アドレス対応表（SPARC内先頭アドレス%g5）の初期値は全エントリ0である．
2. 変換後SPARC命令の初期状態は空である．
3. M命令列が，M内アドレス$2040_{(16)}$（SPARC内アドレス%g4+0x2040）から格納されていると仮定する．
4. モニタが，M命令先頭アドレス$2040_{(16)}$を%g2に設定し，動的変換部が，top:から走行を開始する．
5. %g2を%g3によりマスクした後，左に1ビットシフトし，$2040_{(16)}$番地に対応するアドレス対応表のエントリ（%g5+0x4080）をロードする．
6. エントリの内容が0である場合，変換後SPARC命令は存在しないため，cnv:に分岐し，“M-SPARC命令変換”において，Mの無条件分岐命令（例ではBALR命令）を検出するまで，M命令をSPARC命令に変換する．この際，各M命令に対応する変換後SPARC命令の先頭アドレス（H，I，J，K，L，M，N，O）をアドレス対応表に登録する．4バ

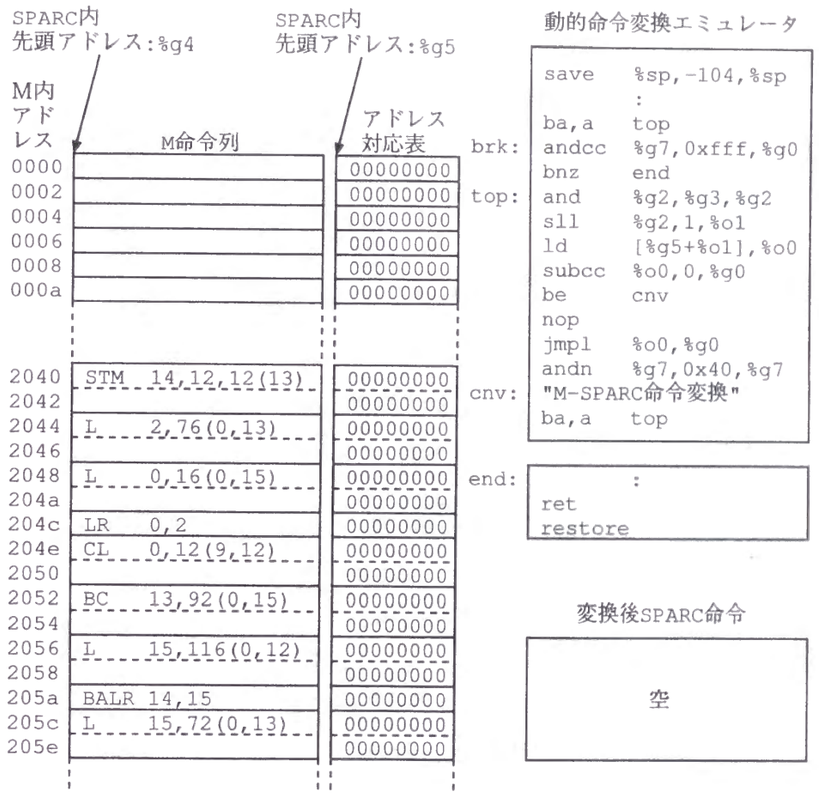


図 4.3: 動的命令変換の初期状態

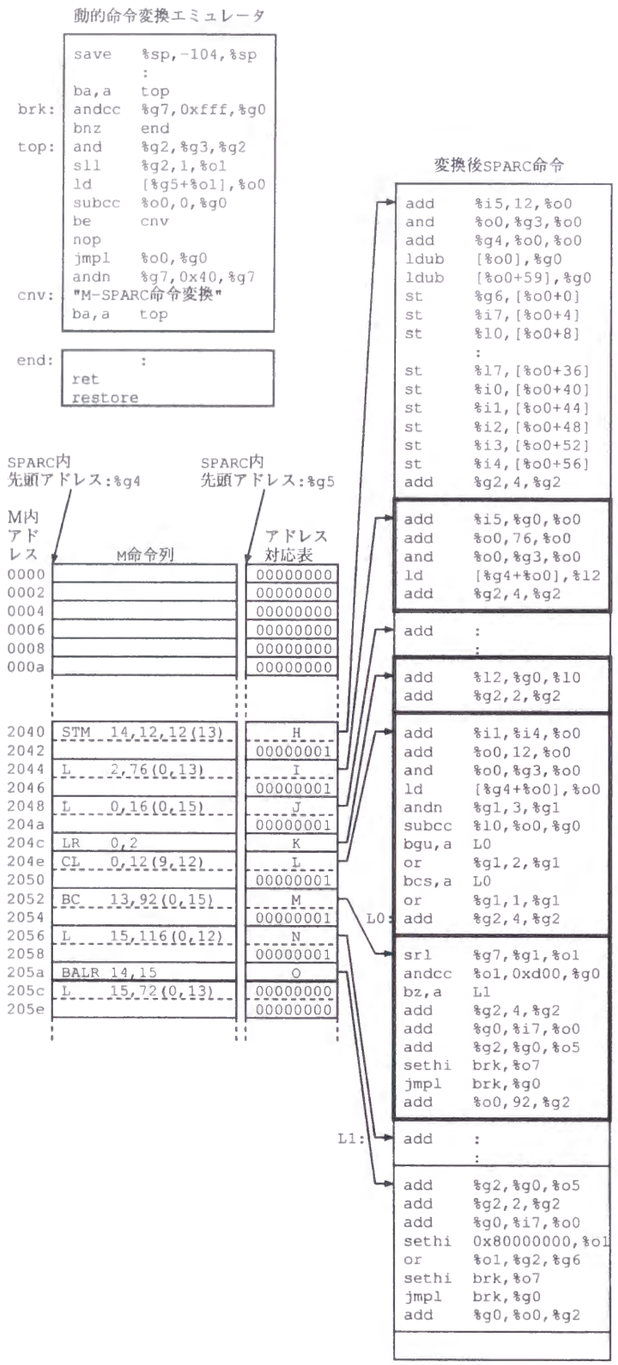


図 4.4: 動的命令変換の走行状態

イト命令および6バイト命令の場合，命令途中に対応するアドレス対応表のエントリには，命令途中を示す$00000001_{(16)}$や$00000002_{(16)}$を格納する．

7. 無条件分岐命令 “ba，a top” により，top:へ戻る．

8. 再度，$2040_{(16)}$番地に対応するアドレス対応表のエントリ（%g5+0x4080）をロードすると，今度はHが格納されているので，無条件分岐命令 “jmpl %o0，%g0” により，変換後SPARC命令へ分岐する．

9. 変換後SPARC命令を実行し，最後に “jmpl brk” により，brk:に戻り，外部割り込みの有無を検査した後，top:すなわち（5）へ戻る．“jmpl brk” 命令は，この例では，BC命令における条件分岐成立時，および，BALR命令の実行終了時に実行する．top:に戻った時，%g2は分岐先アドレスを保持しており，引続き分岐先アドレスの命令を変換し実行する．

4.4.4 例外の検出

本節では，Mにおいて規定される各例外の検出をSPARCによりいかに実現するかについて説明する．例外は，M命令フェッチ時，M命令デコード時，M命令分岐時，Mオペランドフェッチ時，M命令実行時に各々発生する可能性がある．なお，M命令フェッチおよびデコードについては，M命令からSPARC命令に変換する際にのみ行われる．

M命令フェッチ時のページフォルト：

前述したように，Mのアドレス変換には，SPARCのアドレス変換機構を利用している．M命令が属するページが主記憶上に存在しない時，M命令をフェッチするためのSPARCのロード命令がページフォルトを発生する．この時モニタは，OSに対してページフォルトを通知し，OS中の割り込み処理を行うM命令のエミュレーションを開始する．OSによるページインが完了した後，ページフォルトを発生したM命令の実行を再開することにより，M命令のフェッチが完了する．

M命令デコード時の指定例外：

偶数レジスタ番号しか許されていない命令フィールドに奇数レジスタ番号が指定されているなどの指定例外は，命令変換時に検出できる．しかし，実際にその命令が実行されるまでは例外を発生することができないため，例外を発生するだけのSPARC命令に変換する．

M命令分岐時の指定例外：

命令分岐の結果，命令アドレスが奇数となる時には指定例外を検出しなければならない．

これには，M の主記憶アドレス 2 バイト毎に，各 4 バイトのアドレス対応表が対応していることを利用する．アドレス対応表を参照する際には，命令アドレスを 1 ビット左シフトした値をインデックスとする 4 バイトロード命令（ld 命令）を使用している．従って，命令アドレスが奇数である場合には，インデックスが 4 の倍数とならず，ld 命令がアドレス境界違反例外を発生する．このことを利用することにより，アドレスが奇数か否かを明示的に検査することによる性能低下を回避している．アドレス対応表の内部でアドレス境界違反例外を検出した場合，モニタは，OS に対して指定例外を通知する．

変換済み M 命令の途中への分岐:

たとえば分岐先アドレスが 4 バイト長命令の 2 バイト目である場合，M アーキテクチャではそのアドレスから命令が始まるものとして実行する．一方，本システムでも同様の仕様を実現するには，同一アドレスの命令に対して複数の命令解釈を許す必要があり，すなわち，複数組のアドレス対応表を用意しなければならない．しかし，わざわざ命令の途中へ分岐する方法は，非常にトリッキーであり，このためにアドレス対応表の管理が複雑になることは避けたい．従って，本実験システムでは，分岐先アドレスが，既に SPARC 命令に変換された M 命令の途中を指す場合，例外とすることにした．変換後命令の実行開始には，アドレス対応表の内容を分岐先アドレスとする jmpl 命令を使用している．M 命令が，変換済みの M 命令の途中へ分岐する場合，アドレス対応表には，$00000001_{(16)}$ や $00000002_{(16)}$ が格納されているため，jmpl によりこれらのアドレスへ分岐しようとする．jmpl 命令において分岐先が 4 バイト境界でない場合，アドレス境界違反例外が発生する．これを契機としてモニタは，OS に対して例外を通知する．

M オペランドフェッチ時の指定例外:

オペランドアドレスが奇数であるなど，変換後命令の実行時にしか検出できない指定例外は，SPARC 命令により明示的に検査する．

M オペランドフェッチ時のページフォルト:

M 命令フェッチの場合と同様に，M オペランドが属するページが主記憶上に存在しない時，SPARC のロード/ストア命令がページフォルトを発生する．この時モニタは，OS に対してページフォルトを通知し，OS 中の割り込み処理を行う M 命令のエミュレーションを開始する．OS によるページインが完了した後，ページフォルトを発生した M 命令の実行を再開することにより，ロード/ストア命令が完了する．

M 命令実行時の演算例外:

固定小数点演算例外（桁あふれ，除算），浮動小数点演算例外（桁あふれ，下位桁あふれ，除算例外，有効数字），10 進演算例外（桁あふれ，除算，データ）については，SPARC

命令により明示的に検査する．

4.5 詳細構造

本節では，前節において説明した基本的な動作には含まれない，Mアーキテクチャ・エミュレーションにおいて重要な機能について説明する．図4.5に，図4.4に示したM命令列，アドレス対応表，変換後SPARC命令列を含む，データ構造の全体を示す．

4.5.1 アドレス対応表

アドレス対応表は，Mの2バイトごとに4バイトのエントリを占めることから，Mの論理アドレス空間の2倍の空間を必要とする．しかし，実際に必要な領域は，実行する命令アドレスに対応するエントリのみである．このことを利用して，アドレス対応表を節約することが可能である．このためには，SPARCのアドレス変換機構を用いる．アドレス対応表をSPARCの仮想空間上に割り付け，実ページが存在しない状態をアドレス対応表の初期状態とする．動的変換部が，存在しない実ページに属するアドレス対応表のエントリを参照した時に，ページフォルトが発生し，これを契機としてモニタが実ページを割り付け，内容を0に初期化する．

4.5.2 変換後SPARC命令の格納領域

変換後SPARC命令を格納する領域は，以下の条件を満たす必要がある．

- Mのページがページアウトされた場合，そのページに対応する変換後SPARC命令は不要となる．このため，Mの各ページに対応する変換後命令を特定できるようなデータ構造が必要である．
- 後述するように，ストア命令により命令を書き換えながら実行する自己変更プログラムでは，不要となった変換後命令がゴミとして蓄積していく．ゴミを回収するためには，一旦，ゴミおよびゴミ前後の変換後命令を広範囲に消去しなければならない．範囲を容易に特定するためにブロック化が必要である．
- Mには，EXECUTE命令という，オペランドにより指定した命令を間接的に実行する機構がある．オペランドにより指定された命令は，実行するたびにSPARC命令に変換実行され，変換結果は，2度と使用されることはない．従って，EXECUTE命令のオペランドに対する変換後SPARC命令は，他の変換後命令の蓄積に影響を与えない，一時的な場所に格納することが必要である．

これらを満足させるために，変換後SPARC命令を格納する領域は，図4.5に示すように，動的に割り付ける複数のブロックと，静的に割り付けるブロックとに分割して管理している．各ブロックの大きさは32Kバイトである．

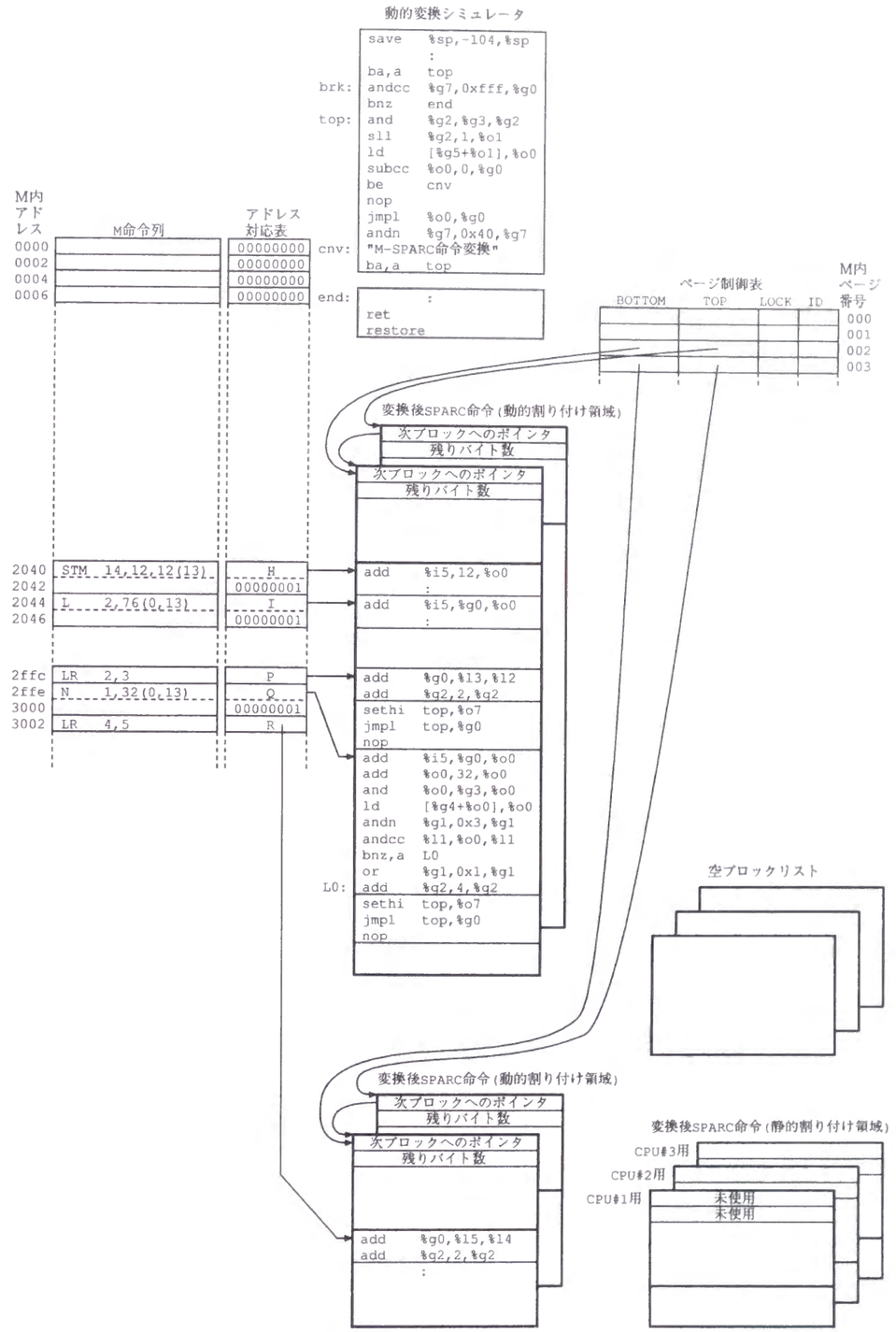


図 4.5: 動的命令変換におけるデータ構造

動的割り付け領域

動的に割り付けるブロックは，変換後SPARC命令を格納する際に，必要に応じて空ブロックリストから割り付け，Mのページごとにエントリを設けたページ制御表からポインタにより繋いでいる．各ブロックの先頭には，次ブロックへのポインタ，自ブロックの残りバイト数，および，その他の制御データを格納し，変換後命令は，先頭から128バイト目以降に格納する．変換後SPARC命令を消去したり，ガベージコレクションを行う際には，Mのページ番号からページ制御表のエントリを求め，継がれているブロックを辿り，各ブロックを順に，空きブロックリストに戻すことにより行う．また，同時に，Mのページに対応するアドレス対応表の各エントリの内容を0に戻す．図4.5のM内アドレス$\texttt{2ffe}_{(16)}$に示すように，M命令がページを跨いでいる場合には，以下の手順に従うことにより，矛盾のない制御を行っている．

- ページ002に対する変換後SPARC命令を消去する場合，$\texttt{2ffe}_{(16)}$に対応するアドレス対応表まで0を書き込む．$3000_{(16)}$に対応するアドレス対応表が，命令途中を示す$00000001_{(16)}$であるので，ここにも0を書き込む．ページ002に対応するブロックを空ブロックリストに戻す．
- ページ003に対する変換後命令を消去する場合，$3000_{(16)}$に対応するアドレス対応表が命令途中を示す$00000001_{(16)}$であるので，さかのぼって，$\texttt{2ffe}_{(16)}$に対応するアドレス対応表から0を書き込む．ページ003に対応するブロックを空ブロックリストに戻す．

静的割り付け領域

静的に割り付けるブロックは，SPARC-CPUごとに1個だけを割り付けており，EXECUTE命令のオペランドにより指定された命令を変換実行する際に，一時的に使用する．

4.5.3 主記憶共有型の複数SPARC-CPU

DS/90が主記憶共有型の複数SPARC-CPUを有する場合，1つのMシステムに対して，複数の動的変換部を動作させるために，M-SPARC命令変換を行う際に，排他制御を行っている．静的割り付け領域は，CPUごとに独立に設けているため，排他制御は必要ない．一方，動的割り付け領域は，複数のCPUが変換後SPARC命令を共有する．Mの同一ページに属する命令を異なるCPUが同時に変換および実行する場合，ページ制御表および動的割り付け領域に矛盾が生じないよう排他制御が必要である．ページ制御表の各エントリごとに1バイトの排他制御情報（LOCK）を設け，アドレス対応表のエントリが0である場合，排他制御情報を用いて，対応するページをロックした後に命令変換を開始する．そのページに関する命令変換が終了した後にロックをはずすことにより，同時には1つのCPUだけがM-SPARC命令変換を行うよう制御している．また，変換後SPARC命令列がtop:へ戻る命令により完結しないうちに，他のCPUが変換後命令の実行を開始しないよう，アドレス対応表への登録は，変換後命令列を完結し，ロックをはずす直前に，アドレスの高い方から低い方へ逆順に行っている．

4.5.4 M-SPARC命令変換の終了条件

これまで部分的に説明した，図4.5における“M-SPARC命令変換”が，M-SPARC命令変換を終了しtop:へ戻るための条件を以下にまとめる．

- 次M命令がMの次ページから開始する場合．(4.5.2節に示した制御のため)
- 次M命令がMの次ページに跨る場合．(4.5.2節に示した制御のため)
- 無条件分岐命令となり得る，BAL，BAS，LPSW，DIAG，EX，SVC命令の変換を終了した時．
- 無条件分岐命令となる，M（条件マスク）=15であるBCおよびBCR命令，また，R2が0でないBALR，BSM，BASSM，BASRの変換を終了した時．
- 次M命令がSPARC命令に変換済（アドレス対応表のエントリが0以外）の時．

4.6 自己変更プログラムのための考慮

Mアーキテクチャは，プログラムが命令語を変更した場合に，命令キャッシュを無効化しなければならないといった特別の手順を義務づけておらず，書き換えられた命令は直ちに有効とならなければならない．このため，ストア命令により命令を書き換えながら実行する自己変更プログラムを正しく動作させるためには，特別の考慮が必要となる．具体的には，(1) 命令語を書き換えたことを確実に検出する手段；(2) M 命令が SPARC 命令に変換された後に M 命令が変更された場合，変換後 SPARC 命令を無効化する手段；が必要である．我々は，以下に示す 2 種類の方法を考案した．

4.6.1 ページに対する書き込み保護機構を利用する方法

SPARC のアドレス変換機構中の書き込み保護機構を利用する方法である．自己変更が一切なく，かつ，M 命令とストアオペランドが同一ページに存在しないという 2 つの条件が満たされる場合には，割り込みが発生せず，動的命令変換本来の性能が発揮できるため，最も望ましい方法である．一方，自己変更が一切無い場合であっても，M 命令とストアオペランドが同一ページに存在する場合は，ストア命令を実行するたびに書き込み保護例外が発生するため，極端な性能低下となる可能性がある．すなわち，ストアオペランドと命令語が同一ページに存在する頻度の大小が，高性能を出せるか否かを決定する重要な要因となる．処理手順を以下に示す．

1. M 命令を SPARC 命令に変換する際に，M 命令アドレスに対応する SPARC ページの書き込み保護機構を有効にする．
2. 以後，M 命令が SPARC 命令に変換されている M ページに対するストア命令は，全て，書き込み保護例外によりモニタに割り込みを発生する．
3. モニタは，例外を検出したオペランドアドレスを使用してアドレス対応表を参照する．内容が 0 である場合には，ストア先が M 命令ではないか，または，変換後 SPARC 命令がまだ存在していないと判断し，処理を終了する．
4. 内容が 0 でない場合には，変更された M 命令の先頭アドレスに対応するアドレス対応表から変換後 SPARC 命令の先頭アドレスを求め，先頭の 1 命令を top:に戻る無条件分岐命令により上書きする．
5. さらにモニタは，変更された M 命令に対応するアドレス対応表の内容を 0 にする．
6. 以上の処理により，変更前の M 命令に対応する変換後 SPARC 命令は無効化され，変更後の M 命令は，次に実行される時に，新たな SPARC 命令に再変換される．

本方法は，ストアオペランドと命令語の配置，すなわちプログラムの性質によって性能が大きく変化することが予想されるため，机上での性能予測は困難である．従って，後述するように，実際にプログラムを走行させて性能測定を行った．

4.6.2 ストア命令がアドレス対応表を検査する方法

M のストア命令をエミュレートする変換後 SPARC 命令が，毎回，オペランドアドレスに対応するアドレス対応表の内容を検査する方法である．割り込みを使用しないため，ストアオペランドと命令語が同一ページに存在する場合でも，性能が大きく低下することはない．しかし，ストアを行う全ての命令の実行速度が一様に低下する．

1. ストア命令に対応する変換後 SPARC 命令が，オペランドアドレスを基にアドレス対応表を参照する．内容が 0 である場合には，ストア先が M 命令ではないか，または，変換後 SPARC 命令がまだ存在していないと判断し，処理を終了する．
2. 内容が 0 でない場合には，変更された M 命令の先頭アドレスに対応するアドレス対応表から変換後 SPARC 命令の先頭アドレスを求め，先頭の 1 命令を top:に戻る命令列により上書きする．
3. さらに，変更された M 命令に対応するアドレス対応表の内容を 0 にする．
4. 以上の処理により，変更前の M 命令に対応する変換後 SPARC 命令は無効化され，変更後の M 命令は，次に実行される時に，新たな SPARC 命令に再変換される．

本方法は，プログラムの性質によって性能が大きく左右されることはない．従って，以下のような机上計算により，性能低下を予測することができる．

まず，ストアを行う命令の出現頻度は，経験上，全命令の 10%程度である．最も一般的な，M の 4 バイトストア命令 ST 2，76（0，13）を実行する際に，アドレス対応表の検査を行うことにすると，以下のような SPARC 命令列となる．なお，アドレス対応表は，M の主記憶 2 バイト毎に 1 エントリ（4 バイト長）が対応するため，ストア先アドレスが 4 バイト境界である場合でも，上位 2 バイトと下位 2 バイトに対応する各々のエントリが 0 であるか否かを検査しなければならない．

```
add   %i5,%g0,%o0   :
add   %o0,76,%o0    : ST 命令に対する
and   %o0,%g3,%o0   : 本来の SPARC 命令列
st    %l2,[%g4+%o0]:
add   %g2,4,%g2     :
```

```
sll   %o0,1,%o1     :アドレスを1bitシフト
add   %g5,%o1,%o2   :対応表の先頭に加算
ld    [%o2],%o3     :上位2バイトに対する
subcc %o3,0,%g0     :   対応表が0以外なら
tnz   SELFMODIFY    :   自己変更有り
ld    [%o2+4],%o3   :下位2バイトに対する
subcc %o3,0,%g0     :   対応表が0以外なら
tnz   SELFMODIFY    :   自己変更有り
```

このように，典型的な4バイトストア命令では，8命令分のSPARC命令が追加される．さて，2バイトストア命令の場合，アドレスが偶数であれば，検査すべきアドレス対応表は1エントリのみなので，前述の例からわかるように5命令の増加で済む．一方，4バイトを超えるストア命令の場合，より多くのエントリを検査しなければならない．このようなばらつきはあるものの，全M命令の10%をストア命令が占めると仮定し，また，各々のストア命令について，平均して8命令分だけSPARC命令が増加すると仮定した場合，M命令を1命令実行するのに要する平均SPARC命令数は，以下のようになる．なお，自己変更を検出しない場合の平均SPARC命令数をSとする．

$$\text{平均命令数} = S \times 0.9 + (S + 8) \times 0.1 \tag{4.1}$$

4.7 各種プログラムを用いた性能評価

本節では，最大3命令を同時実行可能なスーパスカラ・プロセッサであるSuperSPARC（クロックサイクル60MHz，1次命令キャッシュ容量20Kバイト，2次キャッシュ容量1Mバイト）を搭載したFujitsu製DS/90システムを使用して，性能評価を行う．

4.7.1 命令ワーキングセットが小さいプログラムによる評価

まず，命令ワーキングセットが小さく，大部分がプロブレムモードで動作するベンチマークプログラムとして，Dhrystone-2.1およびStanford-integerを取りあげる．これらのプログラムは自己変更を行わないため，自己変更を検出しない動的命令変換手法を使って性能測定および評価を行う．

表4.1に，測定結果を示す．(a) はプログラムの走行所要時間，(b) はエミュレートした全M命令数，(c) は (b) のうちSPARC命令への変換を要したM命令数，(d) はエミュレートのために走行した全SPARC命令数である．

命令数の比（d/b）から，M命令を1命令エミュレートするのに，平均してSPARC命令が約10命令走行することがわかる．これには，条件分岐命令などが前述のbrk:部分を走行するのに要した命令数も含まれる．

SPARC-CPIは，SPARC命令を1命令実行するのに要する平均サイクル数である．ところで，SuperSPARCには，最大3命令を同時実行するスーパスカラ機能を無効化し，命令を1命令ずつ逐次実行させる機構がある．逐次実行させた場合，いずれのプログラムにおいても実行時間が約25%増加した．SuperSPARCの命令レベル並列処理による性能向上は，約25%である．

M-MIPS値（b/a）は，単位時間あたりに実行したM命令数である．(e) は，命令変換のオーバヘッドを測定するために，故意に変換後SPARC命令を蓄積しないよう改造し，M命令を毎回SPARC命令に変換して実行させた場合のプログラムの走行所要時間である．

さて，ここで，M命令を1命令変換するのに要する平均時間を（X），対応する変換後SPARC命令列を実行するのに要する平均時間を（Y）とする．以下では，M命令列を（b）個実行する場合を考える．

SPARC命令を蓄積する場合，変換を要したM命令数（c）を用いると，実行に要する総時間は，以下のように表現できる．

$$\text{実行に要する総時間 } a = X \times c + Y \times b \tag{4.2}$$

一方，SPARC命令を蓄積しない場合，実行に要する総時間は，以下のように表現できる．

表 4.1: 命令ワーキングセットが小さいプログラムの性能

	Dhrystone-2.1 回転数=10000	Stanford-integer
プログラムの実行時間 (a)	1.3 秒	2.9 秒
M 実行命令数 (b)	9.6Mstep	24Mstep
変換を要した M 命令数 (c)	1159step	3228step
再利用率 ((b–c)/b)	99.99%	99.99%
SPARC 実行命令数 (d)	94Mstep	223Mstep
命令数の比 (d/b)	9.8 倍	9.3 倍
SPARC-CPI(a × 60MHz/d)	0.83	0.78
M-MIPS 値 (b/a)	7.4MIPS	8.3MIPS
変換後命令を蓄積しない場合 (e)	16 秒	38 秒
実行時間比 ((e–a)/a)	11 倍	12 倍
1 次キャッシュミス回数/M 命令	0.52 回	0.17 回
2 次キャッシュミス回数/M 命令	0.048 回	0.012 回
ミスアラインのオーバヘッド (f)	0.00%	0.00%

$$\text{実行に要する総時間 } e = X \times b + Y \times b \tag{4.3}$$

さて,(a) における命令変換のオーバヘッドは,(総変換時間) / (総実行時間), すなわち,

$$= \frac{X \times c}{a} \tag{4.4}$$

である. 式 4.2および式 4.3より,

$$= \frac{e-a}{a} \times \frac{c}{b-c} \tag{4.5}$$

$$= \frac{e-a}{a} \times (\frac{b}{b-c} - 1) \tag{4.6}$$

となり, 再利用率 ((b–c) /b) が 99.99%である場合には,

$$= \frac{e-a}{a} \times (\frac{1}{0.9999} - 1) \tag{4.7}$$

$$= \frac{e-a}{a} \times 0.0001 \tag{4.8}$$

となる. 実行時間比 ((e–a) /a) が約 10 であることから, 結局, $10 \times 0.0001 = 0.1\%$となり, 命令変換のオーバヘッドは無視できる. すなわち, 動的命令変換手法はきわめて有効であると言える.

(f) は, SPARC のロード/ストア命令においてアドレス境界違反例外が発生し, モニタが 1 バイトロード/ストア命令を用いて再実行したことによる, プログラムの実行時間 (a) に占めるオーバヘッドである. (f) が 0.00%であることから, アドレス境界違反例外の発生頻度は皆無であり, M の 2 または 4 バイトロード/ストア命令を SPARC の 2 または 4 バイトロード/ストア命令を用いてエミュレートする方法は, 正しいと言える.

4.7.2 実運用状態に近い I/O 頻度を有するジョブを用いた評価

次に, 実運用状態に近い I/O 頻度を有し, 主に OS 性能を測定する FORTRAN プログラムを 10 本同時に走行させた場合の, 性能測定および評価を行う. このプログラムは, 40% が OS 中のスーパバイザモード, 60%がプロブレムモードで走行する.

本プログラムには, 自己変更を行う部分が 1 箇所存在するため, 自己変更を検出する動的命令変換手法を適用しなければならない. 検出する手法としては, 4.6.1節において述べた「ページに対する書き込み保護機構を利用する方法」を実際にインプリメントした.

表4.2に,(1) 自己変更検出有り;(2) 自己変更検出無し;の場合について測定した結果を示す.「1/2次キャッシュミス/M命令」はM命令を1命令実行する際に平均して発生した1/2次キャッシュミスの回数,「モニタ含む/除くMIPS値」は特権命令など,モニタ内のSPARC命令の実行に要する時間を含む/除く場合のMIPS値である.以下では,動的命令変換だけに注目するために「モニタを除くMIPS値」をMIPS値と呼ぶこととし考察を行う.

まず,規模の大きい本プログラムの場合でも,変換後命令再利用率が99.98%ときわめて良好であることが判明した.このような環境においても,命令変換のオーバヘッドを無視することができ,動的命令変換手法における命令蓄積の効果が非常に大きいと言うことができる.同様に,前節において説明したミスアラインのオーバヘッド (f) についても,きわめて小さいと言える.

さて,(1) および (2) の結果から,自己変更を検出しない場合には,2.7MIPSと比較的高い性能が得られるのに対し,自己変更を検出する場合には,0.45MIPSと極端に性能が低下していることがわかる.調査の結果,自己変更のためのオーバヘッドが大きいのではなく,ストアオペランドと命令語が同一ページに存在する頻度が高いために,自己変更ではないにも関わらず,ストア命令を実行するたびに書き込み保護例外が頻発するためであることが判明した.これは,Mのプログラムに関し,命令領域とデータ領域とが明確に分離されるような仕組みが規定されておらず,同一ページ内に命令とデータが共存することが多いことをそのまま反映している.

一方,自己変更を検出するもう一つの方法「ストア命令がアドレス対応表を検査する方法」について評価してみる.4.6.2節の式4.1に,S=10を代入すると,平均SPARC命令数は10.8となる.すなわち,この方法による性能低下は約10%と見積もることができ,前述した「書き込み保護機構を利用する方法」に比べて高速である.

以上のことから,自己変更プログラムについては,「書き込み保護機構を利用する方法」よりも「ストア命令がアドレス対応表を検査する方法」を採用すべきであることが明らかになった.

4.7.3 命令ワーキングセットを変化させた場合の挙動

最後に,Mの命令ワーキングセットサイズを変化させた場合の,性能の変化について測定し考察を加える.

図4.6に,ループを構成するM命令数を横軸,M-MIPS値を縦軸とするグラフを示す.ループ中のM命令列には,BC命令:LA命令:AL命令を1:1:6の比で用いており,1個のM命令あたり,平均13個のSPARC命令に変換される.このうち実際に実行されるSPARC命令は平均10命令である.

この場合,200個のM命令列からなるループは2600個のSPARC命令(命令サイズは10Kバイト)から成るループとなり,500個のM命令列からなるループは6500個のSPARC命令

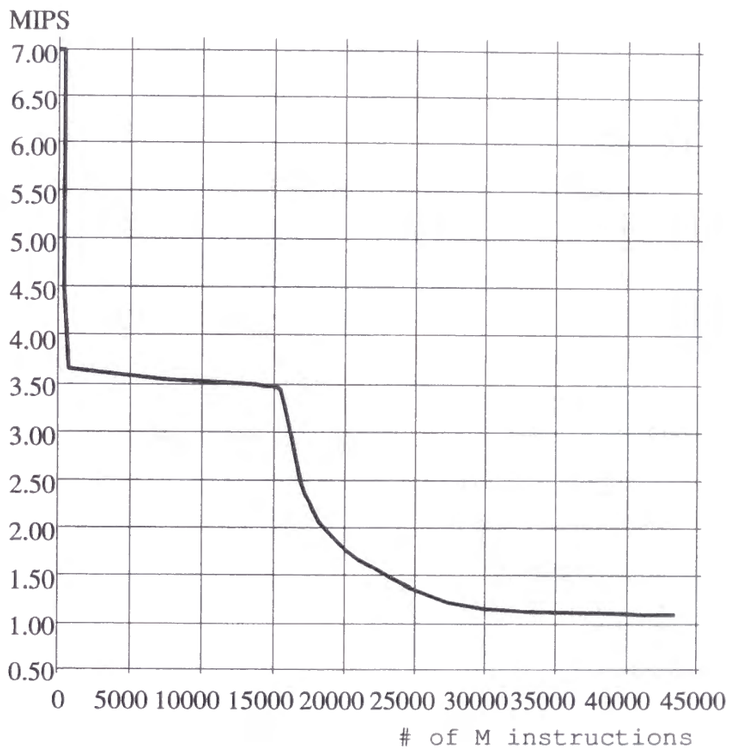


図 4.6: 命令ワーキングセットと性能の関係

（命令サイズは26Kバイト）から成るループとなる．SuperSPARCの1次命令キャッシュ容量は20Kバイトであるので，M命令数約380を境界として1次命令キャッシュミスの影響が出はじめ，M命令数の増加にしたがって約3.5MIPSに性能が低下する．M命令数が15000（変換後SPARC命令数は約195000，命令サイズは780Kバイト）を超えるあたりからは，容量1Mバイトの2次キャッシュにおいてもキャッシュミスが発生しはじめ，性能は1MIPS程度に漸近する．

表4.1および表4.2に示すように，1,2次キャッシュミス回数/M命令が，I/Oジョブ（0.88, 0.14）＞Dhrystone-2.1（0.52，0.048）＞Stanford-integer（0.17，0.012）の順に多くなっている．M命令あたり，SPARCのロード/ストア命令は，1個あるかないか程度の低い頻度でしか出現しないため，M命令あたり0.88～0.17という高い1次キャッシュミス回数の大部分は，命令キャッシュミスで占められると考えることができる．

以上のことから，表4.1に示すように，Dhrystone-2.1およびStanford-integerにおける命令数の比（d/b）がほぼ同じであるにも関わらず，M-MIPS値（b/a）が7.4および8.3と異なること，また，表4.2に示すように，自己変更を検出しない場合のI/OジョブではMIPS値が2.7と低くなるのは，命令キャッシュミス率に大きな差があるためと判断することができる．

このように，動的命令変換手法を用いた場合，変換前のM命令列の特性に比べて，変換後のSPARC命令列の特性は大きく変わる．M命令列は，SPARC命令列に変換されることにより十倍程度の大きさに増大するため，SPARCの命令キャッシュが相対的に小さく見え，命令キャッシュミスによる性能低下をいかに抑えるかが重要な課題となる．

さて，オペランドキャッシュに関しても考察してみる．SPARCのロード/ストア命令として実行されるものには，以下がある．

M命令からSPARC命令への変換時:

M命令をデコードする際のロード，アドレス対応表に対するロード/ストア，SPARC命令を生成する際のロード/ストア

変換後SPARC命令の実行時:

Mのロード/ストア命令を実行するためのロード/ストア命令，主記憶上に設けたMの浮動小数点レジスタに関するロード/ストア

このうち，命令変換のオーバヘッドは無視できるため，Mのロード/ストア命令実行時および浮動小数点レジスタ参照時に発生するロード/ストア命令の出現頻度が，オペランドキャッシュのヒット率に影響を及ぼす．ところが，全体として実行されるSPARC命令数はM命令の十倍程度もあるため，SPARC命令列として見た場合のこれらロード/ストア命令の出現頻度は，M命令列として見た出現頻度の十分の一程度に小さくなる．言いかえると，キャッシュヒット率

は変わらないけれども，ミスペナルティの影響が薄まって見える．このため，SPARCのオペランドキャッシュに関して性能上の大きな問題が生じることはないと言える．

表 4.2: 実運用状態に近い I/O 頻度を有するジョブを用いた評価

	(1) 自己変更検出有り	(2) 自己変更検出無し
全プログラムの実行時間 (a)	1366 秒	457 秒
M 実行命令数 (b)	518Mstep	同左
・内スーパバイザ	193Mstep	同左
・内プロブレム	325Mstep	同左
変換を要した M 命令数 (c)	114Kstep	同左
再利用率 ((b–c)/b)	99.98%	同左
モニタ含む M-MIPS 値	0.35MIPS	1.1MIPS
モニタ除く M-MIPS 値	0.45MIPS	2.7MIPS
1 次キャッシュミス回数/M 命令	1.09 回	0.88 回
2 次キャッシュミス回数/M 命令	0.20 回	0.14 回
ミスアラインのオーバヘッド (f)	0.63%	1.83%

4.8 今後の展望

動的命令変換手法では，変換後命令の実行時に命令キャッシュが相対的に小さく見えること，また，このために，命令キャッシュミスによる性能低下をいかに低減するかがきわめて重要であることが明らかになった．この問題を解決するためには以下の方法が考えられる．これらは今後の課題である．

変換後 SPARC 命令の最適化:

4.2節に示したような，ひと塊の SPARC 命令が M 命令を 1 命令ずつ実行するという現在の命令変換方法を改め，SPARC 命令が数個の M 命令の機能をまとめて実行するよう最適化する方法が考えられる．この場合，(1) 従来，M 命令を 1 命令実行する毎にプログラムカウンタを更新していたのに対して，複数 M 命令を実行後に，まとめて更新することができる；(2) 後続する M 命令が，それまでの条件コードを参照することなく，条件コードを上書きする場合，先行する M 命令では条件コードの生成を省略することができる；という効果により，SPARC 命令を削減することができると考えられる．ただし，プリサイス・インタラプトを保証するためには，途中で例外が発生した場合に，つじつまが合うよう，SPARC 命令の実行順序を考慮する必要がある．また，自己変更を検出した場合には，前後の M 命令に対応する変換後命令を含めて，広範囲に変換後命令を無効化しなければならない．さらに，広範囲な最適化を行うほど命令変換に要する時間が大きくなり，最適化の程度と命令変換オーバヘッドとのトレードオフを慎重に検討する必要があるといった問題点がある．最適化の効果は期待できるものの，多くの問題を解決しなければならない．

ソフトウェアによる命令のプリフェッチ:

命令キャッシュへのプリフェッチをソフトウェアが制御できるようなプロセッサの場合，変換後 SPARC 命令に適当な間隔でプリフェッチを行わせることにより，命令キャッシュミスによる性能低下を軽減することが考えられる．変換後 SPARC 命令には，ロード/ストア命令と同様に，分岐命令についても出現頻度が低いという特徴が見られる．すなわち，基本ブロックが比較的大きいことを利用して，たとえば，基本ブロックの先頭において，次の基本ブロックの命令をプリフェッチしておくといった工夫により，命令キャッシュミスによる性能低下を比較的容易に抑えることができると考えている．

4.9 おわりに

本章では，Mアーキテクチャ上で動作するオペレーティング・システムを動的命令変換手法によりSPARCシステム上で動作させる実験システムについて，動作原理を詳述し，ベンチマーク・プログラムを走行させて得た結果に基づき詳細な性能評価を行った．

まず，M命令を1命令エミュレートするのに，平均してSPARC命令が約10命令走行することを確認した．変換後SPARC命令の再利用率がきわめて高く，かつ，命令変換に要する時間が変換後SPARC命令のみの実行に要する時間の約10倍に過ぎないことから，命令変換のオーバヘッドは無視できるほど小さく，動的命令変換手法がきわめて有効であることを明らかにした．

また，40%という高い割合でOS中の命令が走行するプログラムの場合でも，変換後命令再利用率が99.98%ときわめて良好であり，変換後命令の蓄積の効果が非常に大きく，本手法が有効であることを明らかにした．

さらに，自己変更プログラムについて，動的命令変換手法により高い性能を得るためには，「書き込み保護機構を利用する方法」よりも「ストア命令がアドレス対応表を検査する方法」を採用すべきであることを明らかにした．

一方，動的命令変換手法ではM命令のワーキングセットの大きさに依存して，SPARCの命令キャッシュヒット率が敏感に変化すること，すなわち，変換後命令が変換前に比べて十倍程度の大きさに増大するため，SPARCの命令キャッシュが相対的に小さく見えること，従って，命令キャッシュミスによる性能低下をいかに抑えるかが重要であることを明らかにした．

最後に，命令キャッシュミスによる性能低下を低減し，さらに性能を向上させるための方策に触れ，(1) 変換後SPARC命令の最適化；(2) ソフトウェアによる命令のプリフェッチ；を挙げた．

Chapter 5

結論

本論文では，「超高速プロセッサの構成方式とエミュレーションに関する研究」と題し，最近注目を集めている，長形式命令語方式，および，命令エミュレーション方式について，実際にシステム開発を行い，それぞれ，詳細な評価を行った．

長形式命令語方式を採用した，VPP500スカラプロセッサでは，限られたハードウェア資源の中でいかに高性能を引き出すかの工夫を施し，従来のVP2600スカラよりも遅いサイクルタイムであるにも関わらず，同等の性能を達成した．特に，従来のMアーキテクチャに対して，大幅にアーキテクチャを改善することにより，インプリメント上の工夫による高速化の可能性を大きく広げた．実際に，いくつかのベンチマークプログラムを走行した結果から，アーキテクチャおよびインプリメントが，コンパイラ技術とうまくかみ合っており，長形式命令語方式および非同期実行機構が，きわめて有効に機能することがわかった．長形式命令語方式は，今後，性能を向上するための手段として，1命令語あたりの操作数を増やし，ハードウェアの並列度を増すだけでなく，より多くの投機的実行を可能とするための機構として，(1) 条件分岐を越える命令スケジューリングを容易にするためのアシスト機構；(2) 効果的なキャッシュ・プリフェッチ機構；を装備することがきわめて重要であると考えている．

次に，命令エミュレーション方式を採用した，Mアーキテクチャでは，動的命令変換手法を採用し，オペレーティング・システムを含めた，全てのソフトウェア資産を，SPARCシステム上で動作させる実験システムを開発した．測定の結果，命令変換のオーバヘッドは無視できるほど小さく，動的命令変換手法がきわめて有効であることを明らかにした．今後，さらなる高性能化を図るためには，変換後命令の最適化を行う必要がある．ところで，変換後命令の最適化技術は，変換前の命令列における冗長な命令をも検出し，変換前の命令列にさかのぼって，最適化を行うことができる可能性を秘めている．実は，このような最適化は，長形式命令語方式やスーパスカラ方式において，高性能を達成するためのコンパイラ技術と同じものである．プロセッサを構成するにあたって，いかに，コンパイラ技術が重要であるかを，改めて認識す

るものである．

さて，これまでは，同じアーキテクチャの中で，ソフトウェアとハードウェアが機能分担を行い，コストと性能を最適にする努力が払われてきた．同じアーキテクチャを仮定し，ソフトウェアおよびハードウェアが互いに高性能化を図ることは，性能の面から見た場合，最も有効なアプローチである．特に，性能を重視する超高速計算機システムについては，今後も，ベクトル化や並列処理による高性能化が追求され，今後も発展していくであろう．

一方，現在まで，多くのソフトウェア資産を有するアーキテクチャこそが，市場を支配することができると考えられてきた．しかし，エミュレーション技術の進歩により，これからは，ソフトウェア資産とアーキテクチャは，それほど密接に関連しなくなる時代が来ると確信している．エミュレーション技術による，ソフトウェア資産の移植性の向上により，ソフトウェアとハードウェアを自由に組み合わせることができるようになる．この結果，優れたソフトウェアとハードウェアが，お互いのアーキテクチャに束縛されることなく，より自由に機能や性能を向上することができる方向に進むであろう．このことは，これから新しいプロセッサを開発しようとする技術者に対して，大きなチャンスを与えるとともに，従来，豊富なソフトウェア資産を武器として，高い市場シェアを誇ってきたプロセッサにとっては，大きな脅威となることを意味している．このような，自由な競争は，特に，ネットワーク・コンピューティングの分野では，標準的な技術になると考えている．機種に依存しない中間コードにより記述されたプログラムは，ネットワークを経由して，様々なアーキテクチャのシステム上で動作することができる．一旦は影を潜めてしまった，中間コードという概念が，再び注目されることになるであろう．

パーソナルコンピュータやワークステーションの性能が急激に向上するにつれて，分散処理を指向したシステムが，数多く構築されてきた．しかし，結局，分散できるのは，演算処理能力だけであり，データは，集中的に管理されてこそ，使いやすいのである．今後，計算機システムは，巨大なデータベースや計算資源を集中管理する高速な計算機システムと，アーキテクチャに縛られることなく，様々なソフトウェアを実行することのできる，マルチメディア指向の端末計算機システムという，2 極分化の方向に進むことにより，これまで以上に，自由な，かつ，激しい開発競争が行われるようになると考えている．

序論において述べたように，いずれの方向においても，ソフトウェアの時間的/空間的局所性をハードウェアに伝達し，ハードウェアがこれを有効に活用することこそが，計算機システムの性能を向上させる根源である．われわれ計算機技術者は，ソフトウェアおよびハードウェアのいずれにも偏ることなく，継続性と発展性の両方を兼ね備えた計算機システムを開発していかなければならない．

謝辞

本論文の執筆にあたり，京都大学の富田眞治教授には，さまざまな貴重な御助言を賜わった．ここに深甚なる謝意を表したい．

さて，本論文をまとめることができたのは，在学中から，現在にいたるまで，数多くの方々から，御指導，御助言を頂いたおかげである．すべての方々のお名前を列挙することはできないが，ここに深く感謝したい．

在学中は，特に，萩原宏 京都大学名誉教授，柴山潔 京都工芸繊維大学教授，新實治男 京都工芸繊維大学助教授に御指導を賜わった．ここに深謝したい．

富士通株式会社では，青木隆 第一コンピュータ事業部長，菊池伸行 主席部長，二野井栄三 第二技術部長から，本論文の執筆に関して御理解を頂き，いろいろ便宜を図っていただいた．ここに謝意を表したい．

VPP500 スカラプロセッサに関する研究では，田村秀夫 第一開発統括部担当部長，北村俊明課長，竹部好正氏をはじめとする多くの方々から，さまざまな御指導，御助言，御協力を頂いた．また，M アーキテクチャの命令エミュレーションに関する研究では，上埜治彦課長，鈴木貴朗氏，田尻邦彦氏をはじめとする方々と，貴重な議論を重ね，また，数多く助けて頂いた．ここに，謝意を表したい．

最後に，妻 志保と，いつも笑顔で励ましてくれた息子 秀樹に感謝する．

参考文献

[1] 富田眞治: “並列計算機構成論”, 昭晃堂, 1986.

[2] 富田眞治: “コンピュータアーキテクチャ”, 丸善, 1994.

[3] Patterson D. A., Hennessy J. L.: “Computer Architecture : A Quantiative Approach”, Morgan Kaufmann, 1990. (邦訳) 富田眞治, 村上和彰, 新實治男: “コンピュータ・アーキテクチャ ー 設計・実現・評価の定量的アプローチ”, 日経 BP 社, 1992.

[4] Uchida N., Hirai M., Yoshida M., Hotta K.: “Fujitsu VP2000 Series”, Digest of Papers, COMPCON Spring 90, pp. 4-11, Feb. 1990.

[5] IBM: “IBM システム/370 拡張アーキテクチャー解説書”, SA22-7085-0.

[6] Miura K., Takamura M., Sakamoto Y., Okada S.: “Overview of the Fujitsu VPP500 Supercomputer”, Digest of Papers, COMPCON Spring 93, 1993.

[7] Utsumi T., Ikeda M., Takamura M.: “Architecture of the VPP500 Parallel Supercomputer”, Proceedings of Supercomputing ’94, pp. 478-487, Washington D.C., Nov. 1994.

[8] Nakanishi M., Ina H., Miura K.: “A High Performance Linear Equation Solver on the VPP500 Parallel Supercomputer”, Proceedings of Supercomputing ’94, pp. 803-810, Washington D.C., Nov. 1994.

[9] Iwashita H., Okada S., Nagakura H.: “VPP Fortran and Parallel Programming on the VPP500 Supercomputer”, Proceedings for Poster Sessions of the International Symposium of Parallel Architectures, Algorithms, and Networks ’94, Kanazawa, Japan, Dec. 1994.

[10] 中島康彦, 北村俊明, 田村秀夫, 滝内政昭: “VPP500 スカラプロセサの特徴”, 情報処理学会研究報告, ARC-104-17, pp. 129-136, Jan. 1994.

[11] Nakashima Y., Kitamura T., Tamura H., Takiuchi M., Miura K.: "Scalar Processor of the VPP500 Parallel Supercomputer", Proceedings of 9th ACM Int. Conf. of Supercomputing, pp. 348-356, Jul. 1995.

[12] 中島康彦, 大野優人, 竹部好正: "VPP500 スカラプロセサの性能", 情報処理学会論文誌, Vol. 38, No. 4, pp. 863-872, 1997.

[13] David Callahan, Ken Kennedy, Allan Porterfield: "Software Prefetching", Proceedings of ASPLOS-IV, pp. 40-52, Apr. 1991.

[14] Moudgill M., Vassiliadis S.: "Precise Interrupts", IEEE MICRO, pp. 58-67, Feb. 1996.

[15] "SPEC Newsletter", Vols. 1-6, 1989-1994.

[16] Giladi R., Ahituv N.: "SPEC as a Performance Evaluation Measure", IEEE Computer, pp. 33-42, Aug. 1995.

[17] Mirghafori N., Jacoby M., Patterson D.: "Truth in SPEC Benchmarks", Computer Architecture News, Vol. 23, No. 5, Dec. 1995.

[18] Allan V. H., Jones R. B., Lee R. M., Allan S. J.: "Software Pipelining", ACM Computing Surveys, Vol. 27, No. 3, Sep. 1995.

[19] Allan V. H., Shah U. R., Reddy K. M.: "Petri Net versus Modulo Scheduling for Software Pipelining", Proceedings of MICRO-28, pp. 105-110, 1995.

[20] Fisher J. A.,: "Trace scheduling: A technique for global microcode compaction", IEEE Trans. on Computers, Vol. 30, pp. 478-490, Jul. 1981.

[21] Nicolau A.: "Percolation Scheduling: A Parallel Compilation Technique", Computer Sciences Technical Report 85-678, Cornel Univ., May. 1985.

[22] Lowney P. G., Freudenberger S. M., Karzes T. J., Lichtenstein W. D., Nix R. P., O'Donnell J. J., Ruttenberg J. C.: "The Multiflow trace scheduling compiler", The Journal of Supercomputing, Vol. 7, pp. 51-142, Jan. 1993.

[23] Dehnert J. C., Hsu P. Y.-T., Bratt J. P.: "Overlapped loop support in the Cydra-5", Proceedings of ASPLOS-III, pp. 26-38, Apr. 1989.

[24] Huff R. A.: "Lifetime-sensitive modulo scheduling", SIGPLAN Programming Language and Design Implementation, pp. 258-267, Jun. 1993.

[25] Mahlke S. A., Lin D. C., Chen W. Y., Hank R. E., Bringmann R. A.: "Effective compiler support for predicated execution using the hyperblock", Proceedings of MICRO-25, pp. 45-54, Dec. 1992.

[26] Rau B. R., Schlansker M. S., Tirumalai P. P.: "Code generation schema for modulo scheduled loops", Proceedings of MICRO-25, pp. 158-169, Dec. 1992.

[27] Rau B. R., Lee M., Tirumalai P. Schlansker M.: "Register allocation for software pipelined loops", Proceedings of SIGPLAN'92, pp. 283-299, Jun. 1992.

[28] Rau B. R., Fisher J. A.: "Instruction-level parallel processing: History, overview, and perspective", Supercomputing 7, pp. 9-50, 1993.

[29] Rau B. R., Yen D. W. L., Yen W., Towle R. A.: "The Cydra-5 departmental supercomputer: Design philosophies, decisions, and trade-offs", IEEE Computer, pp. 12-25, Jan. 1989.

[30] Tirumalai P., Lee M., Schlansker M. S.: "Parallelization of loops with exits on pipelined architectures", Proceedings of SuperComputing'90, pp. 200-212, Nov. 1990.

[31] Warter N. J., Haab G. E., Bockhaus J. W.: "Enhanced modulo scheduling for loops with conditional branches", Proceedings of MICRO-25, pp. 170-179, Dec. 1992.

[32] Warter J. J., Partamian N.: "Modulo Scheduling with Multiple Initiation Intervals", Proceedings of MICRO-28, pp. 111-118, 1995.

[33] Ramakrishnan S.: "Software pipelining in PA-RISC compilers", Hewlett-Packard Journal, pp. 39-45, Jul. 1992.

[34] Natarajan B., Schlansker M.: "Spill-Free Parallel Scheduling of Basic Blocks", Proceedings of MICRO-28, pp. 119-124, 1995.

[35] Eichenberger A. E., Davidson E. S.: "Stage Scheduling: A Technique to Reduce the Register Requirements of a Modulo Schedule", Proceedings of MICRO-28, pp. 338-349, 1995.

[36] Chang P. P., Warter N. J., Mahlke S. A, Chen W. Y., Hwu W. W.: "Three Architectural Models for Compiler-Controlled Speculative Execution", IEEE Trans. on Computers, Vol. 44, pp. 481-494, Apr. 1995.

[37] Hwu W. W., Patt Y. N.: "Hpsm, a high performance restricted data flow architecture having minimal functionality", Proceedings of 13th Int'l Symp. Computer Architecture, pp. 297-306, Jun. 1986.

[38] Smith M. D., Lam M. S., Horowitz M. A.: "Boosting beyond static scheduling in a superscalar processor", Proceedings of 17th Int'l Symp. Computer Architecture, pp. 344-354, May. 1990.

[39] Smith M. D., Johnson M., Horowitz M. A.: "Limits on multiple instruction issue", Proceedings of ASPLOS-III, pp. 290-302, Apr. 1989.

[40] Colwell R. P., Nix J. J., O'Donnell J. J., Papworth D. B., Rodman P. K.: "A VLIW architecture for a trace scheduling compiler", Proceedings of ASPLOS-II, pp. 180-192, Apr. 1987.

[41] Ando H., Nakanishi C., Hara T., Nakaya M.: "Unconstrained Speculative Execution with Predicated State Buffering", Proceedings of ISCA'95, pp. 126-137, 1995.

[42] 安藤秀樹, 中西知嘉子, 原 哲也, 中屋雅夫: "プレディケーティング:VLIW マシンにおける投機的実行のためのアーキテクチャ上の支援", 情報処理学会論文誌, Vol. 37, No. 11, pp. 2039-2055, 1996.

[43] 安藤秀樹, 中西知嘉子, 原 哲也, 中屋雅夫: "VLIW マシンのための非数値計算応用向き広域命令スケジューリング手法", 情報処理学会論文誌, Vol. 38, No. 9, pp. 1812-1828, 1997.

[44] Mahlke S. A., Hank R. E., McCormick J. E., August D. I., Hwu W. W: "A Comparison of Full and Partial Predicated Execution Support for ILP Processors", Proceedings of ISCA'95, pp. 138-149, 1995.

[45] Chow P., Horowits M.: "Architecture tradeoffs in the design of MIPS-X", Proceedings of the 14th Annual International Symposium on Computer Architecture, Jun. 1987.

[46] Melear C.: "The design of the 88000 RISC family", IEEE MICRO, pp. 26-38, Apr. 1989.

[47] Lee J., Smith A.: "Branch Prediction Strategies and Branch Target Buffer Design", Computer, 17(1), Jan. 1984.

[48] Smith J. E.: "A Study of Branch Prediction Strategies", Proceedings of the 8th International Symposium on Computer Architecture, Jun. 1981.

[49] Yeh T. Y., Patt Y. N.: "Two-level Adaptive Branch Prediction", 24th ACM/IEEE International Symposium on Microarchitecture, Nov. 1991.

[50] McFarling S.: "Combining Branch Predictors", WRL Technical Note TN-36, Digital Equipment Corp., Jun. 1993.

[51] Yeh T. Y., Patt Y. N.: "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History", 20th Annual International Symposium on Computer Architecture, May. 1993.

[52] Gloy N., Young C., Chen J. B., Smith M. D.: "An Analysis of Dynamic Branch Prediction Schemes", Proceedings of ISCA'96, pp. 12-21, 1996.

[53] Sechrest S., Lee C. C., Mudge T.: "Correlation and Aliasing in Dynamic Branch Predictors", Proceedings of ISCA'96, pp. 22-32, 1996.

[54] 小松 秀昭, 古関 聰, 鈴木秀俊, 深澤 良彰: "拡張 VLIW プロセッサ GIFT における命令レベル並列処理機構", 情報処理学会論文誌, Vol. 34, No. 12, pp. 2599-2610, 1993.

[55] 小松 秀昭, 古関 聰, 深澤 良彰: "命令レベル並列アーキテクチャのための大域的コードスケジューリング技法", 情報処理学会論文誌, Vol. 37, No. 6, pp. 1149-1161, 1996.

[56] Young C., Gloy N., Smith M. D.: "A Comparative Analysis of Schemes for Correlated Branch Prediction", Proceedings of ISCA'95, pp. 276-286, 1995.

[57] Young C., Smith M. D.: "Improving the Accuracy of Static Branch Prediction Using Branch Correlation", Proceedings of ASPLOS-VI, pp. 232-241, Oct. 1994.

[58] Pan S., So. K., Rahmeh J.: "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", Proceedings of ASPLOS-V, Oct. 1992.

[59] Calder B., Grunwald D.: "Reducing Branch Costs via Branch Alignment", Proceedings of ASPLOS-VI, pp. 242-251, Oct. 1994.

[60] Lesartre G., Hunt D.: "PA-8500:The Continuing Evolution of the PA-8000 Family", Digest of Papers, COMPCON Spring 97, 1997.

[61] Patterson J. R. C.: "Accurate Static Branch Prediction by Value Range Propagation", Proceedings of SIGPLAN'95, pp. 67-78, 1995.

[62] Uhlig R., Nagle D.: "Instruction Fetching: Coping with Code Bloat", Proceedings of ISCA'95, pp. 345-356, 1995.

[63] Lee D., Baer J., Calder B., Grunwald D.: "Instruction Cache Fetch Policies for Speculative Execution", Proceedings of ISCA'95, pp. 357-367, 1995.

[64] Austin T. M., Pnevmatikatos D. N., Sohi G. S.: "Atreamlining Data Cache Access with Fast Address Calculation", Proceedings of ISCA'95, pp. 369-380, 1995.

[65] Chen T. F.: "An Effective Programmable Prefetch Engine for On-Chip Caches", Proceedings of MICRO-28, pp. 237-242, 1995.

[66] Chen T. F.: "Effective Hardware-Based Data Prefetching for High-Performance Processors", IEEE Trans. on Computers, Vol. 44, No. 5, pp. 609-623, May. 1995.

[67] Lipasti M. H., Schmidt W. J., Kunkel S. R., Roediger R. R.: "SPAID: Software Prefetching in Pointer- and Call-Intensive Environments", Proceedings of MICRO-28, pp. 231-236, 1995.

[68] Santhanam V., Gornish E. H., Hsu W. C.: "Data Prefetching on the HP PA-8000", Proceedings of the International Symposium on Computer Architecture (ISCA'97), Jun. 1997.

[69] Ozawa T., Kimura Y., Nishizaki S.: "Cache Miss Heuristics and Preloading Techniques for General-Purpose Programs", Proceedings of MICRO-28, pp. 243-236, 1995.

[70] 小沢年弘, 西崎慎一郎, 木村康則: "ロード命令の先行実行とその評価", 情報処理学会研究報告, ARC-109-1, pp. 1-8, Dec. 1994.

[71] 小沢年弘, 西崎慎一郎, 木村康則: "ロード命令の先行実行の科学技術計算プログラムにおける評価", 情報処理学会研究報告, ARC-112-4, pp. 25-32, Jun. 1995.

[72] Conte T. M., Sathaye S. W.: "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures", Proceedings of MICRO-28, pp. 208-218, 1995.

[73] Tremblay M., O'Connor J. M., Narayanan V., He L.: "VIS Speeds New Media Processing ", IEEE Micro, pp. 10-20, Aug. 1996.

[74] Peleg A., Weiser U.: "MMX Technology Extension to the Intel Architecture", IEEE Micro, pp. 42-50, Aug. 1996.

[75] Lee R. B.: "Subword Parallelism with MAX-2", IEEE Micro, pp. 51-59, Aug. 1996.

[76] IBM: "Enterprise Systems Architecture/390 Principles of Operation", SA22-7201-02.

[77] Jayakumar M., McCalla T.: "Simulation of Microprocessor Emulation Using GASP-PL/I", Computer, Vol. 10, No. 4, pp. 20-26, 1977.

[78] Mallach E.: "Emulator Architecture", Computer, Vol. 8, No. 8, pp. 24-31, 1975.

[79] Benjamin R.: "The Spectra 70/45 Emulator for the RCA 301", CACM, Vol. 8, pp. 748-752, 1965.

[80] McCormack A., Schansman T., Womack K.: "1401 Compatibility Feature on the IBM System/360 Model 30", CACM, Vol. 8, pp. 287-297, 1965.

[81] Schoen T., Belsole M.: "A Burroughs 220 Emulator for the IBM 360/25", IEEE Transactions on Computers, Vol. C-20, pp. 795-798, 1971.

[82] Tucker S.: "Emulation of Large Systems", CACM, Vol. 8, pp. 753-761, 1965.

[83] Digital Equipment Corporation: "Freeport Express white papers", FreePort Express Information center, 1996, http://www.service.digital.com/freeport-express/

[84] Hohensee P., Myszewski M., Reese D.: "Wabi CPU Emulation", Proceedings of HOT Chips 8, Stanford Univ., Aug. 1996.

[85] SPARC International: "The SPARC Architecture Manual Version 8"

[86] SPARC International: "The SPARC Architecture Manual Version 9"

[87] Cohn R., Lowney P.: "Hot Cold Optimization of Large Windows/NT Applications", Proceedings of MICRO-29, pp. 80-89, 1996.

[88] Tremblay M., O'Connor M.: "PicoJava: A hardware Implementation of the Java Virtual Machine", Proceedings of HOT Chips 8, Stanford Univ., Aug. 1996.

[89] Hsieh C., Gyllenhaal J., Hwu W.: "Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results", Proceedings of MICRO-29, pp. 90-99, 1996.

著者発表論文

論文

1. 中島康彦, 新實治男, 柴山潔, 萩原宏: “3 次元形状モデリングにおける立体集合演算の並列処理方式”, 情報処理学会論文誌, Vol. 30, No. 10, pp. 1298-1308, 1989.

2. 中島康彦, 大野優人, 竹部好正: “VPP500 スカラプロセサの性能”, 情報処理学会論文誌, Vol. 38, No. 4, pp. 863-872, 1997.

3. 中島康彦, 上埜治彦, 田尻邦彦, 鈴木貴朗: “動的命令変換手法による M アーキテクチャ・エミュレーション”, 情報処理学会論文誌, Vol. 38, No. 11, pp. 2309-2320, 1997.

国際会議

1. Nakashima Y., Kitamura T., Tamura H., Takiuchi M., Miura K.: “Scalar Processor of the VPP500 Parallel Supercomputer”, Proceedings of 9th ACM Int. Conf. of Supercomputing, pp. 348-356, Jul. 1995.

口頭発表

1. 中島康彦, 新實治男, 富田眞治, 萩原宏: “実時間 3 次元動画システムにおける動画記述”, 第 33 回情報処理学会全国大会論文集, 2Q-2, pp. 2073-2074, 1986.

2. 中島康彦, 北村俊明, 田村秀夫, 滝内政昭: “VPP500 スカラプロセッサの特徴”, 情報処理学会研究報告, ARC-104-17, pp. 129-136, Jan. 1994.